# API Protocol Compliance in Object-Oriented Software

**Kevin Bierhoff**

23 April 2009
CMU-ISR-09-108

Institute for Software Research
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

**Thesis Committee:**
Jonathan Aldrich, chair
David Garlan
Peter Lee
Alex Aiken (Stanford University)

*Submitted in partial fulfillment of the requirements*
*for the degree of Doctor of Philosophy*



This work was supported in part by DARPA grant #HR00110710019, NSF grants CCF-0811592 and CCF-0546550, Army Research Office grant #DAAD19-02-1-0389 entitled "Perpetually Available and Secure Information Systems", the Department of Defense, NASA cooperative agreements NCC-2-1298 and NNA05CS30A, and the Software Industry Center at CMU and its sponsors, especially the Alfred P. Sloan Foundation. The views and conclusions of this dissertation are those of the author and do not necessarily reflect the views of any funding agencies or the U.S. government.

# Report Documentation Page

*Form Approved*
*OMB No. 0704-0188*



| 1. REPORT DATE | 2. REPORT TYPE | 3. DATES COVERED |
|---|---|---|
| **23 APR 2009** | | **00-00-2009 to 00-00-2009** |

| 4. TITLE AND SUBTITLE | 5a. CONTRACT NUMBER |
|---|---|
| **API Protocol Compliance in Object-Oriented Software** | 5b. GRANT NUMBER |
| | 5c. PROGRAM ELEMENT NUMBER |
| 6. AUTHOR(S) | 5d. PROJECT NUMBER |
| | 5e. TASK NUMBER |
| | 5f. WORK UNIT NUMBER |
| 7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **Carnegie Mellon University,School of Computer Science,Institute for Software Research,Pittsburgh,PA,15213** | 8. PERFORMING ORGANIZATION REPORT NUMBER |
| 9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) | 10. SPONSOR/MONITOR'S ACRONYM(S) |
| | 11. SPONSOR/MONITOR'S REPORT NUMBER(S) |

12. DISTRIBUTION/AVAILABILITY STATEMENT
**Approved for public release; distribution unlimited**

13. SUPPLEMENTARY NOTES

14. ABSTRACT
**see report**

15. SUBJECT TERMS

| 16. SECURITY CLASSIFICATION OF: | | | 17. LIMITATION OF ABSTRACT | 18. NUMBER OF PAGES | 19a. NAME OF RESPONSIBLE PERSON |
|---|---|---|---|---|---|
| a. REPORT **unclassified** | b. ABSTRACT **unclassified** | c. THIS PAGE **unclassified** | **Same as Report (SAR)** | **158** | |

**Keywords:** Typestate, access permissions, state refinement, Plural.

**Abstract**

Modern software development is highly reliant on reusable APIs. APIs often define *usage protocols* that API clients must follow in order for code implementing the API to work correctly. Loosely speaking, API protocols define legal sequences of method calls on objects. In this work, protocols are defined based on *typestates* (Strom and Yemini, 1986; DeLine and Fähndrich, 2004b). Typestates leverage the familiar intuition of abstract state machines to define usage protocols.

The goal of this work is to give developers comprehensive help in defining and following API protocols in object-oriented software. Two key technical contributions enable the proposed approach: (1) Object state spaces are defined with *hierarchical state refinements*. Hierarchical state spaces make specifications more succinct, elegantly deal with subtyping, express uncertainty, and enable more precise reasoning about aliasing. (2) A novel abstraction, called *access permissions*, combines typestate and aliasing information. Access permissions capture developers' design intent regarding API protocols and enable sound modular verification of API protocol compliance while allowing a great deal of flexibility in aliasing objects.

This dissertation demonstrates that *typestate-based protocols with state refinement and access permissions can be used for automated, static, modular enforcement of API protocols in practical object-oriented software.* Formal and empirical results show that the presented approach captures common API protocols succinctly, allows sound modular checking of protocol compliance in object-oriented code, can be automated in tools for mainstream programming languages that impose low annotation burden on developers, and can check API protocols in off-the-shelf software with higher precision than previous approaches.

This work puts automatic API protocol compliance checking within reach of being used in practice. It will enable rapid and correct use of APIs during initial construction and ensure that API clients and implementations remain consistent with the specified protocol during maintenance tasks.

*For my parents,*
*who have given me ability and encouragement to pursue an academic career,*
*who have always supported me in my quest,*
*and who have been there for me whenever I needed their help.*
*Ich liebe Euch sehr.*

# Acknowledgments

This work would not have been possible without the help, encouragement, and support of many people.

I thank my advisor, Jonathan Aldrich, for his continuous advice and support all along. I also thank the other members of my thesis committee, David Garlan, Peter Lee, and Alex Aiken, for their advice and engagement.

Furthermore, I am deeply indebted to the many people who over the years have provided feedback, input, and encouragement. Thanks to all of you. I want to thank Nels Beckman in particular for being both a great collaborator and friend, and my friends George Fairbanks and Ciera Jaspan for all their help and support. In addition to my committee members, I was fortunate to get advice from many members of the Carnegie Mellon faculty, including Bob Harper, Jim Herbsleb, Mary Shaw, and Frank Pfenning, as well as several "outsiders" including John Boyland, Manuel Fähndrich, Viktor Kuncak, and Rustan Leino. Many others at Carnegie Mellon have challenged, helped, and encouraged me over the years, including Sebastian Boßung, Kevin Bowers, Shang-Wen Cheng, Neel Krishnaswami, Donna Malayeri, Sven Stork, Joshua Sunshine, William Lovas, Tom Murphy, Michael Richter, Rob Simmons, the members of my research group and the Software Systems Study Group, and too many more to mention here. You all have made this work into what it is.

Last but not least I want to thank my family, friends, and fellow students for their support and for making my years at Carnegie Mellon a fun and enriching experience. I thank Maja for her patience with me and her invaluable support. Brianne, Ciera, Dominique, Dorothee, Edith, Elke, George, Ivonne, Kai, Lucia, Hans, Maja, Marc, Michael, Nels, Owen, Robert, Saul, Sebastian, Tian, Tom, Verena, William, Wolfgang, and all the others that I am forgetting: thank you for a wonderful time. I hope that you all continue to be part of my life.

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

## 1.1 APIs and Object Protocols

Modern software development is highly reliant on reusable *APIs* (Application Programming Interfaces). Many programming languages in industrial use such as Java or C# include enormous "standard" libraries for everyday tasks, and organizations often use additional libraries and frameworks that were developed in-house, in the open-source community, or by third parties. For example, the Java Database Connectivity API (JDBC) defines canonical interfaces for accessing relational databases from Java programs using SQL queries which developers can use without having to worry about the details of accessing the particular database they employ.

Reusable APIs often define *usage protocols*. Loosely speaking, usage protocols impose constraints on the order in which events are allowed to occur. For example, a database connection can only be used to execute SQL queries while it is open. Running an SQL query will return a "result set" object that can be used to iterate over the rows (tuples) of the query result using the `next` operation (Figure 1.1). This operation will return `true` if the result set was advanced to a *valid* row and `false` otherwise. Column values can be retrieved from *valid* rows, but attempts to retrieve column values in the *end* state violate the protocol for `ResultSet`. Furthermore, result sets can only be used until the connection they were created on is closed.[1]

It has been a long-standing research challenge to ensure *statically* (before a program ever runs) that API protocols are *followed in client programs* using an API. A long-overlooked but related problem is to make sure that the protocol being checked is consistent with what the *API implementation*, such as implementations of JDBC interfaces for a particular database, does. Both of these challenges are complicated by object *aliasing* (objects being referenced and possibly updated from multiple places in the program)—the hallmark feature of imperative languages like C and Java.

The goal of this work is to give developers comprehensive help in following API protocols as well as to allow them to correctly and concisely formalize protocols for their own code. Our studies of APIs, primarily from the Java standard library, show that challenges in this area revolve around *expressiveness*, *subtyping*, *inheritance*, and *aliasing*. These challenges will be described in more detail below.

This dissertation proposes a set of techniques that address these challenges. The techniques can formalize and statically enforce API protocols in common object-oriented programs. Case studies show that these

[1] This description is simplified: connections and result sets are related through intermediary "statement" objects (Section 7.1.1).

techniques can express interesting and relevant protocols of APIs defined in the Java standard library, and that the tools we developed can automatically verify compliance to these protocols in open-source software codebases.

### 1.1.1 Helping Developers

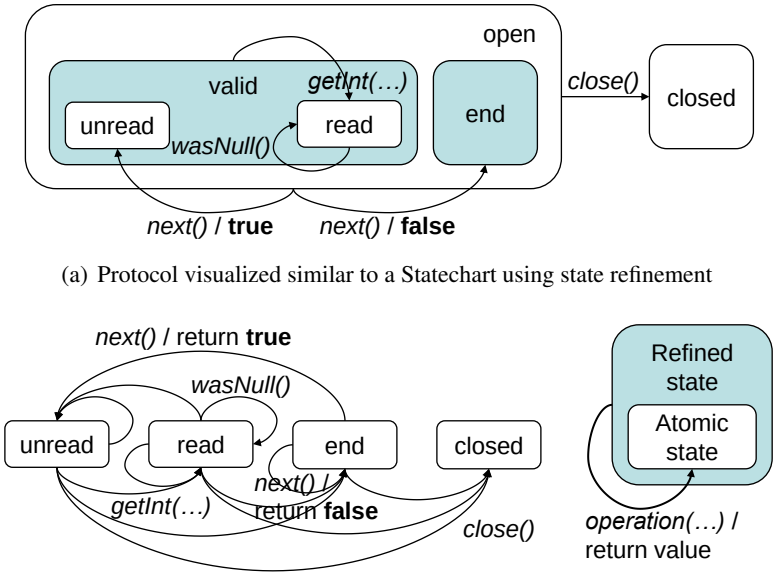
Currently, it is a painstaking and time-consuming undertaking for a developer to gain familiarity with an API. Developers typically create a series of (hopefully small) programs that exercise the API to understand how it works, for instance by invoking operations whose names promise to be relevant for the task at hand. If the developer is fortunate then the API implementation will throw meaningful runtime exceptions when protocols are violated. But at other times, API invocations may simply not produce the expected result, i.e., they silently fail, or protocol violations may corrupt internal data structures of the API implementation, leading to subtle errors *later* (when these data structures are accessed again), or even security flaws exploitable by an attacker. In these cases, developers may be forced to use the debugger to investigate the API internals. If available, the developer can also read documentation, example code, online forums, or the API implementation code, which may accelerate or decelerate the process depending on the quality of these artifacts.[2] What we need is a situation where developers can write code against an unfamiliar API and tools will point out when they start violating protocols, allowing developers to focus on the task they are trying to accomplish.

Even familiar API protocols can be difficult to follow in all cases. Protocols of real APIs are complex and easily violated. In particular, if objects created through the API depend on each other in some way, are stored in fields—or are otherwise shared—then ensuring compliance to API protocols becomes a non-local problem that is challenging even for expert developers. Moreover, if a program behaves as expected, there is no guarantee that API protocols will be followed on all paths through the program. This reduces developers' confidence in their code, especially during maintenance tasks, which can lead to long testing cycles and high costs to fix bugs late in the development process. We would like to quickly check that protocols are consistently followed when code (or its protocol) is written or changed.

Producing reusable APIs is time-consuming as well because protocols have to be correctly documented and implementations have to comply with their documented protocols. In order to avoid erratic behavior at runtime, API implementations have to protect themselves against protocol violations by clients, requiring consistent and expensive runtime checks on method arguments. If data structures are shared between API clients and implementation then API implementation code has to be prepared for client modifications of these data structures. Reentrancy and potential overriding further exacerbate the challenges for API developers. Here, we would like to check that "bad states" are avoided and implementations are consistent with documented protocols. Moreover, if clients are trustworthy, we could potentially speed up programs by removing defensive runtime checks.

**Using this work.** For the purpose of this work, we roughly distinguish API *designers*, API *client developers*, and API *implementers*. Reflecting upon the scenarios described above, our goals include aiding API designers in *documenting* API protocols and client developers in *understanding* APIs; to *assure* to client

[2]The author had an experience with the Eclipse IDE's "builder" infrastructure where (a) names used in the API did not reflect their meaning, (b) the documentation did not state important rules, and (c) sample code included with Eclipse was faulty (fixed upon request by the author with bug 263807). Several hours of using the debugger finally shed light on the problem.




(a) Protocol visualized similar to a Statechart using state refinement

(b) Flat state machine

(c) Legend

Figure 1.1: Simplified JDBC result set protocol. Return values for `next` are included to show the possible outcomes of dynamic state tests.

developers that API protocols are followed in their code; and to help API implementers create *dependable* API implementations. In order to be applicable to large codebases, we would like to, as much as possible, check protocols *automatically* (instead of relying on manual proofs) and in a way suitable for *interactive* use (in order to support developers while they write or maintain code). The work presented in this dissertation accomplishes these goals as follows:

- **Documentation and understanding.** API designers can use hierarchical state machines, such as the one illustrated in Figure 1.1(a), to succinctly describe API protocols. Additionally, they use "access permissions" to describe how API methods will access and use method parameters. Hierarchical state machines and access permissions also address a number of other concerns and are described in more detail below.

  Figure 6.1 shows a description of the `ResultSet` protocol discussed in this chapter using the annotations defined by our tool, Plural. There, Java annotations `@Full` and `@Pure` indicate modifying and reading access to the receiver object, respectively, and specify states required or ensured by the annotated methods, directly encoding the protocol shown in Figure 1.1(a). The conventional Java type signature for `ResultSet` is unchanged by these annotations.

  These annotations allow API designers to concisely and formally *document* their protocols; they can also help developers using an API *understand* how the API works, even without tool involvement.

- **Automation, assurance, and interactivity.** Our tool, Plural, *automatically* checks at compile-time whether a given API client program complies with API protocols described as above. Plural works

with conventional Java code, and similar tools could be developed for other programming languages. Plural determines for every variable at every program point what is known about the state of the referenced object. This information is known as the object's current "typestate" (Strom and Yemini, 1986) and allows *assuring* that API protocols are followed.

Plural may need additional annotations for method parameters and instance fields to check individual methods in a API client program independently from the rest of the program. Our case studies show that this approach requires only about one developer-provided annotation per method in an API client program. These additional annotations enable quick checking of individual methods (currently around 100ms) and allows making assumptions about unfinished or unavailable parts of the program.

API client developers can use Plural *interactively* while they write code in their development environments. Figure 6.2 shows a small `ResultSet` client with an annotation for a method parameter. This annotation allows making all calls on the annotated parameter, *rs*, that are legal according to its protocol (Figure 6.1). Plural also detects a protocol violation. When the developer fixes the problem she can re-run Plural to assure that the modified code respects all API protocols.

- **Dependability.** Separately, developers of API implementations can use Plural to make sure that their code is consistent with the published protocol (see Section 3.2).

This work takes the point of view that a protocol is part of an API's interface that should be *captured* as part of that interface and subsequently can be *enforced* (Section 6.6 summarizes specific use cases of the tool). This improves encapsulation of APIs because protocols can be captured abstractly without specific knowledge of implementation details. It also improves their reusability because API clients can be assured that they use the API according to its protocol.

### 1.1.2 Challenges

The approach presented in this dissertation is based on existing work on *typestates* (Strom and Yemini, 1986). The idea behind typestates is to express protocols, like the `ResultSet` protocol discussed above, as state machines (Figure 1.1). Such protocols allow tracking (statically or dynamically) the current state of each object. As such, typestates are an intuitive mechanism for describing protocols which can be enforced automatically in a sound and modular fashion (DeLine and Fähndrich, 2001, 2004b). Typestates are particularly appealing for object-oriented software because typestates characterize objects abstractly, without revealing implementation details (DeLine and Fähndrich, 2004b). Subtyping, however, creates challenges for typestate-based approaches.

For this thesis we studied five commonly used APIs from the Java standard library including the Java Collections and Database Connectivity (JDBC) APIs (Bierhoff and Aldrich, 2005, 2007b; Bierhoff et al., 2009).[3] This section summarizes the challenges we found with expressing and enforcing protocols for these APIs using typestates, including expressiveness, subtyping and inheritance, and aliasing. The next section will provide an overview of how these challenges are addressed in this work.

**Expressiveness.** Real APIs define complex protocols, and it is crucial that developers be able to express their protocols; otherwise, we cannot check code for compliance to these protocols. As discussed in more

[3] Four of them are subject of case studies presented in Section 7.1.

detail in Section 7.4.1, we found three recurring patterns (which appeared in at least 3 of the 5 APIs we studied) of protocols which are not sufficiently expressible in previous protocol enforcement approaches. One of them, *dependent objects*, is not readily checkable with any automated modular program verification approach (such as Flanagan et al., 2002; Barnett et al., 2004) of which we are aware. For example, dependent objects can be found in the JDBC API: multiple queries can be run on a single connection to the database, resulting in multiple result sets. While result sets are in use, they require the connection to remain open, i.e., `ResultSet` objects *depend on* the `Connection` object they were created on.

Moreover, flat state machines quickly suffer from state explosion problems when trying to describe the intricacies of API protocols, making them tedious to manually define and understand (Bierhoff and Aldrich, 2005). For example, we found 52 distinct states when specifying the Java standard library class `PipedInputStream` (Bierhoff and Aldrich, 2005). To be comprehensible, specifications of such protocols should remain succinct, a common challenge with approaches based on state machines that is not addressed in previous typestate-based techniques.

**Subtyping and inheritance.** Subtyping and inheritance are hallmark features of object-oriented programming. Subtypes should respect *behavioral subtyping* (Liskov and Wing, 1994) and remain *substitutable* for supertypes. However, typestates have traditionally impeded substitutability (Bierhoff and Aldrich, 2005): subtypes often need additional states to describe their protocols, which prohibits objects from being substituted when they are in states unknown in supertypes. For instance, we distinguish `open` and `closed` states for `Connection` objects, but not for other types of objects. Previous approaches would forbid substituting an open `Connection` for an `Object` (its supertype) because `Object` only defines a "default" state that is not the same as `open` (DeLine and Fähndrich, 2004b).

Behavioral subtyping requires *protocol inheritance*: protocols defined in supertypes have to be respected when using subtypes. Subtypes may, however, want to concretize an inherited imprecise protocol or react to new input, which is not allowed in previous typestate-based approaches (Bierhoff and Aldrich, 2005).

Finally, when sub*classes* override a method in previous approaches, they cannot choose whether they want to invoke the overridden method or not—that choice is made by the superclass (DeLine and Fähndrich, 2004b). This limits opportunities for reuse and creates problems for subclasses such as Java's `BufferedInputStream` which only *sometimes* invokes overridden methods (details in Section 3.2.2).

**Automated sound and modular protocol checking under aliasing.** Most software written in imperative and object-oriented programming languages crucially relies on aliasing. Aliasing means that a single object (or memory location) is referenced from multiple places in the program. We will often say that these references "alias" the commonly referenced object.

Aliasing greatly complicates the ability of developers and tools alike to ensure statically that a program module complies with API protocols. This is because aliasing makes reasoning about program behavior a non-local problem: aliasing is typically used to give different parts of a program access to the same object.

It turns out that aliasing is not only common in imperative programs, but is also inherent to many APIs. For instance, the *dependent objects* pattern mentioned above can be seen as a particular form of aliasing where result sets alias the connection they were created on. However, the connection will typically *also* be aliased from the client program that created connections and result sets. The trouble now is to ensure that the client program closes the connection only after closing all the result sets that depend on it.

Aliasing also makes it more difficult to check whether a class implements its specified protocol because reentrant callbacks through aliases can lead to unexpected state changes. For instance, we can trigger the equivalent of a buffer overrun in Java's `BufferedInputStream` through reentrant callbacks (Bierhoff and Aldrich, 2007a). By reentrancy we mean that a method executes within the dynamic scope of the same or another method defined in the same class and with the same receiver object.

Existing automated static protocol checking approaches fall into two categories. Some protocol checkers are whole-program (global) analyses, i.e., they check an entire codebase at once. Whole-program analyses typically account for aliasing, but they can be imprecise, they cannot analyze incomplete programs, and they are typically too slow to be used interactively by a developer.

The other category of protocol checkers is modular. Modular protocol checkers can support developers while they write code: like a typechecker, they check each method separately for protocol violations while assuming the rest of the system to behave as specified. The trade-off has been that modular checkers require code to follow pre-defined patterns of aliasing. Generally speaking, state changes are only allowed where the checker is sure that the changing object is not accessed through other references, simplifying sound reasoning about the states of referenced objects.

This approach of largely forbidding aliasing has serious drawbacks. First, many examples of realistic code are excluded. Moreover, from a developer's point of view, the boundaries of what a checker supports might not fit with the best implementation strategy for a particular problem. Finally, aliasing restrictions arguably leave developers alone just when they have the most trouble in reasoning about their code, namely, in the presence of subtle aliasing. This is exactly the case for JDBC: aliasing restrictions prevent existing modular checking approaches from handling multiple result sets over a single connection, although using JDBC is tricky precisely because of possible interference through aliases.

Unfortunately, it is difficult to offer programmers the flexibility to alias objects in their code while maintaining *precise sound* and *modular* reasoning about protocol compliance. *Soundness* means that we will not miss potential protocol violations in any execution of the program.[4] With *modularity* we mean that we can analyze a part of the program (a single method in our implementation) for protocol compliance. Finally, *precision* means keeping the number of false positives small. (False positives are protocol violation warnings that do not result in actual protocol violations at run time.) Since we are dealing with an in general statically undecidable problem, avoiding both false negatives and false positives is impossible, but a precise sound analysis avoids false negatives completely and keeps false positives to a minimum.

Unrestricted aliasing defies precise, sound, modular reasoning because the presence of other references to objects manipulated in the current module allows the possibility of these objects changing state at virtually any moment while code from the current module executes, *even when the program runs sequentially in a single thread*. In particular, when analyzing the code in a given method, objects could change state as a result of every method call in the code. Soundness would require "forgetting" the state of all objects after every method call, which would lead to a very imprecise analysis. Not forgetting the state of potentially aliased objects will result in a more precise, but grossly unsound, analysis. Achieving both soundness and precision in a modular analysis is exceedingly difficult in the presence of aliasing.

[4] Errors related to race conditions are beyond the scope of this dissertation. However, this work can be soundly extended to concurrent programs (Beckman et al., 2008).

## 1.2 This Dissertation

The research presented in this dissertation addresses the challenges we outlined above. There are two key technical innovations enabling the work presented in this dissertation, *state refinement* and *access permissions*. We briefly introduce them below before we discuss how they help dealing with API protocols.

- **State refinement.** Instead of using a flat state space with a set of mutually exclusive states, in this work, state spaces are hierarchical. Sets of mutually exclusive states *refine* more coarse-grained states, possibly along different orthogonal *dimensions*, making protocol specifications akin to Statecharts (Harel, 1987). This concept makes specifications more succinct, elegantly deals with subtyping, and enables more-precise reasoning about aliasing. Additionally, state refinement can be used to encode *uncertainty* (for instance, which state a result set is in after a call to `next`), which is useful in cases where APIs make *internal choices*.
- **Access permissions** are an abstraction that combines typestate with aliasing information about objects (Bierhoff and Aldrich, 2007b). Developers use access permissions to express the *design intent* of their protocols in annotations on methods and classes. For every reference, access permissions flexibly delineate the effects of aliasing, allowing protocol checking to be appropriately conservative in reasoning about that reference. The primary mechanisms for constraining aliasing effects are to restrict aliases to be *read-only*, i.e., to not modify state, and to make references respect a *state guarantee*, i.e., to not leave a certain state. These mechanisms achieve high precision in tracking effects of possible aliases.

The following subsections discuss in detail how state refinement and access permissions address the challenges discussed above (expressiveness, subtyping and inheritance, and aliasing). Afterwards we summarize the contributions of this dissertation. Section 3 will discuss the approach in detail.

### 1.2.1 Expressiveness

#### Expressive specifications

It is crucial that developers be able to express their protocols. Our approach is able to capture challenging API protocol patterns in common use. In particular, dependencies between objects can be expressed with permissions, and uncertainty can be captured by referring to refined states. In addition, we employ a logic for expressing complex protocols with disjunctions and implications. In practice, we only needed this power in very stylized ways, namely for "dynamic state tests" (such as the `next` method in Figure 1.1, whose return value "indicates" or implies that the result set is in a certain state), "method cases" (which can be seen as conjunctions between function types, see Dunfield and Pfenning (2004)), and for recovering access to fields of dead objects (which we model with a linear implication).

#### Succinct specifications

Protocols of realistic interfaces quickly become complex (Bierhoff and Aldrich, 2005; Bierhoff, 2006). To be comprehensible, specifications of such protocols should remain succinct.

In our approach, "new" typestates are introduced by *refining* an existing one into a set of more fine-grained states. This idea corresponds to OR-states in Statecharts (Harel, 1987).

State refinement allows more-succinct specifications in many cases. For instance, it allows the `ResultSet` protocol from the previous section to be specified with 5 state transitions. The same protocol requires 12 transitions in a flat state machine (Figure 1.1).

By allowing states to be refined multiple times the proposed approach avoids state explosion in many cases. We allow multiple orthogonal refinements of the same typestate. Such orthogonal *state dimensions* correspond to AND-states in Statecharts (which are used to specify parallel state machines) and reduce the number of distinguished states (Bierhoff and Aldrich, 2005). Likewise, the overhead for specifying methods can be further reduced because specifications only need to concern themselves with state dimensions relevant to the method's behavior (see Section 3.1).

### 1.2.2 Subtyping and Inheritance

#### Subtype substitutability

Subtyping and inheritance are hallmark features of object-oriented programming. Subtyping requires objects of subtypes to be *substitutable* for supertypes. State refinement as introduced above helps with subtype substitutability: typestates defined in supertypes are inherited and can be refined by subtypes. Unlike with existing approaches, this guarantees substitutability because every typestate of a subtype has a (possibly less precise) corresponding typestate in its supertypes.

#### Behavioral subtyping

It is a common desideratum in behavioral specifications that subtypes be able to change protocols defined in supertypes. In particular, a subtype might want to define a more-precise protocol than required by its supertypes, or react to additional inputs. Our approach allows such protocol changes as long as *behavioral subtyping* (Liskov and Wing, 1994) is preserved, i.e., the new protocol is guaranteed to be compatible with the original protocol. Behavioral subtyping is ensured by checking that specifications of overriding methods logically imply specifications of overridden methods.

#### Flexible overriding

Inheritance is handled in a novel way, giving subclasses flexibility in method overriding. Subclasses can choose whether and when they want to call overridden methods. This is necessary for reasoning about realistic examples of inheritance such as Java's `BufferedInputStream` (details in Section 3.2.2).

### 1.2.3 Aliasing Flexibility

Aliasing is a significant complication in checking whether clients observe a protocol: a client does not necessarily know whether a reference to an object is the only reference that is active at a particular execution point. This also makes it more difficult to check whether a class implements its specified protocol because reentrant callbacks through aliases can again lead to unexpected state changes.

This dissertation contributes a sound modular typestate verification approach that allows a great deal of flexibility in aliasing. For each reference, it tracks the extent of possible aliasing, and is appropriately conservative in reasoning about that reference. This helps developers account for object manipulations that

may occur through aliases. High precision in tracking effects of possible aliases together with systematic support for "dynamic state tests", i.e., runtime tests on the state of objects, make this approach feasible.

Precise alias tracking is achieved with a novel abstraction, access permissions, that combines typestate with aliasing information about objects. Developers use access permissions to express their protocols in annotations on methods and classes. A modular checking approach verifies that code follows declared protocols (both on the client and provider side of an API).

Access permissions are associated with object references ("pointers") and govern access to the referenced object. We distinguish exclusive (unique), exclusive modifying (full), read-only (pure), immutable, and shared access (Table 1.1). Furthermore, permissions include a *state guarantee*, a state that the method promises not to leave (Bierhoff and Aldrich, 2007b). For example, `next` can promise not to leave the *open* state (Figure 1.1). Reasoning with permissions has the flavor of rely-guarantee reasoning (Giannakopoulou et al., 2004) because permissions allow making assumptions about the rest of the program which are then validated by putting restrictions on other permissions.

Permissions capture three kinds of information:

1. *What kinds of references exist?* We distinguish read-only and modifying access separately through the current and all other references to the same object, leading to the five kinds of permissions shown in Table 1.1.

2. *What state is guaranteed?* References can rely on the guaranteed state even if the referenced object could be modified by other references.

3. *What is known about the current state of the object?* In order to enforce protocols, we need to keep track of what state(s) the referenced object is currently in. Knowledge about the current state at least includes the guaranteed state but can be more specific.

Permissions can only co-exist if they do not violate each other's assumptions. Thus, the following aliasing situations can occur for a given object: a single reference (unique), a distinguished writer reference (full) with many readers (pure), many writers (share) and many readers (pure), and no writers and only readers (immutable and pure).

Permissions are linear—i.e., they cannot be duplicated—in order to preserve consistency between permissions for the same object. But unlike linear type systems (Wadler, 1990), they allow aliasing. This is because permissions can be *split* when aliases are introduced. For example, we can split a unique permission into a full and a pure permission, written $\mathsf{unique} \Rrightarrow \mathsf{full} \otimes \mathsf{pure}$, to introduce a read-only alias. Using *fractions* (Boyland, 2003) we can also *merge* previously split permissions when aliases disappear (e.g., when a method returns). This supports recovering a more powerful permission. Fractions are rational numbers between 0 and 1 expressing how often a permission was split. For example, $\mathsf{full} \Rrightarrow \frac{1}{2} \cdot \mathsf{share} \otimes \frac{1}{2} \cdot \mathsf{share} \Rrightarrow \mathsf{full}$.[5] Splitting a permission into two means replacing it with two new permissions whose fractions sum up to the fractions in the permission being replaced. Merging two permissions does the opposite.

Access permissions offer programmers flexibility to alias objects in their code while maintaining precise sound and modular reasoning about protocol compliance when these objects are used. The ability to analyze programs in a modular way is crucial for at least four reasons:

[5] In previous work, fractions below one make objects immutable (Boyland, 2003); in our approach, they can alternatively indicate shared modifying access, as in this example.

| Access through other permissions | Current permission has ... Read/write access | Read-only access |
|---|---|---|
| None | unique (Boyland, 2003) | – |
| Read-only | full (Bierhoff, 2006) | immutable (Boyland, 2003) |
| Read/write | share (DeLine and Fähndrich, 2004b) | pure (Bierhoff, 2006) |

Table 1.1: Access permission taxonomy

1. It simplifies scaling to large programs because the program can be analyzed in small chunks.

2. It enables analyzing a program module independently from its clients *and* independently from the libraries and frameworks the module relies upon. This allows our approach to check compliance of API implementations to their declared protocols. In contrast, whole-program analyses cannot check an API implementation independently from its clients.

3. It also is crucial for using our approach in practical settings, where different modules of a program are provided by different groups of developers. In many cases, developers may not even have access to all the source code in a project. In these situations, our approach can check protocol compliance in one module independently from the others.

4. Modularity makes our approach akin to a conventional typechecker, allowing developers to check protocols (automatically) while they perform development and maintenance tasks.

As detailed in the previous section, unrestricted aliasing defies precise, sound, modular reasoning because the presence of other references would require "forgetting" the state of all objects after every method call, which would lead to a very imprecise analysis. Permissions obviate or limit the need to forget states, leading to high reasoning precision in many cases. Therefore, our approach balances aliasing restrictions with reasoning precision in a novel way that permits aliasing patterns which could not be analyzed precisely in a modular fashion before.

### 1.2.4 Contributions

This dissertation demonstrates that state refinement and access permissions can express interesting protocols of real object-oriented APIs and can enforce these protocols in practical object-oriented code. Contributions of this dissertation include the following:

1. The approach improves the *expressiveness* of typestate-based protocol specifications for capturing protocols commonly occurring in practice. For example, in the result set protocol discussed above, the `next` method can transition to two different states (Figure 1.1). This *internal choice* can be captured with this approach but not with existing typestate-based approaches. Our approach can also capture *dependent object* patterns, which are insufficiently supported in existing modular program verification approaches (see below).

2. The approach makes typestates interact more seamlessly with *subtyping and inheritance* than previous typestate-based and other static protocol enforcement techniques. In particular, sub*types* are guaranteed to be substitutable and can define new protocols, as long as they remain compatible with protocols defined in supertypes, and sub*classes* are free to use or not use code inherited from superclasses.

3. The approach offers *flexibility in dealing with aliasing*, i.e., the presence of multiple references to the same object, while maintaining modularity and soundness in protocol checking. Previous modular protocol checkers only allow state changes when all references to an object are known; when a checker loses track of a reference (sometimes called an "escaping alias") then no further state changes are allowed. The approach presented in this dissertation supports such escaping aliases while maintaining modularity in checking. This enables protocol enforcement in many examples of API protocols, including result sets over database connections, that could not be verified in a modular fashion before.

4. The presented protocol checking approach can be *automated* in practical tools for conventional programming languages and therefore is available directly to developers. Common API protocol patterns can be supported with such tools. We provide a prototype tool, Plural, that checks API protocols in Java. It is based on a inference system for *polymorphic* fractional permissions, i.e., it supports quantification over exactly how often permissions were split before reaching a given program point, facilitating modular protocol specifications.

5. *Applicability to practical software.* Case studies show that the approach can express interesting API protocols occurring in practice with moderate overhead. On average, we needed about 2 annotations per API method to define protocols. Case studies with open-source programs show that we can also check compliance to these protocols with sufficient precision (less than 1 false positive in 100 LOC), low annotation overhead (about 1 annotation per method), and sufficient performance for interactive use (about 100ms per method on average). One of our case studies suggests that our approach can check protocol compliance in many situations that cause imprecisions in state-of-the-art whole-program protocol analyses (Naeem and Lhoták, 2008; Bodden et al., 2008).

While aliasing flexibility is important for many reasons, our approach enables expressing a common API protocol pattern that to our knowledge is not readily expressed in any previous modular program verification approach: *dependent objects*. This pattern describes situations in which several objects depend on another object. Sometimes it is just one object that depends on another, such as a buffered "wrapper" stream around another stream (cf. Section 3.2), in which case one could use ownership (e.g. Barnett et al., 2004) to model such dependencies. But ownership does not help when multiple objects depend on a single "server" object. For example, all result sets depend on the connection they were created on to remain open as long as result sets are used. Notice that we do not have one result set depend on "its" connection, which would be expressible with ownership, but many.

We can model these dependencies as aliasing *inherent in the API*: An object is aliased from multiple other objects, all of which are defined in the API. As an added benefit, we can allow objects to be aliased *both* from other objects defined in the API and by API clients. For example, with permissions, clients can use statements to run SQL queries and also access the connection directly—as long as they do not close it. Naeem and Lhoták (2008) have pointed out that existing typestate-based approaches (including, in their assessment, this work) cannot express such protocols. In fact, no existing automated modular program verification approach that we are aware of can readily express these protocols. As Chapter 7 shows, we have

been successful in encoding and enforcing the dependencies we found (objects remaining in a given state and objects being immutable) with access permissions.

## 1.3 Potential Impact

This work puts API protocol enforcement within reach of software engineering practice. The work allows building automated tools, such as the one described in this dissertation, for helping developers deal with real API protocols in practical software. This work enables rapidly and correctly using APIs during both software construction and maintenance. APIs will become more dependable as a result of checking their implementations against their declared protocols. Finally, evolution of API clients and implementations during maintenance tasks is simplified.

Furthermore, the intuition of state machines is widely used to specify behavior, and notably in specifying embedded systems. Therefore, this work might be another step along the path of establishing a rigorous connection between systems specifications and implementations.

The author dares to speculate that a time of Pluralism—"Permissions Let Us Reason about ALiasing In a Static Manner"—is upon us. Fractional permissions have already received a great deal of attention in the context of enforcing race freedom (Boyland, 2003; Terauchi and Aiken, 2008) and separation logic (Bornat et al., 2005). This dissertation proposes a broader set of permissions for fine-grained aliasing control and shows that permissions allow precise, sound, and modular reasoning about program behavior while providing aliasing flexibility. Furthermore, permissions can express common aliasing patterns that cannot be captured with state-of-the-art automated program verification techniques, which are based on ownership (Bierhoff and Aldrich, 2008), and could therefore be useful for program verification in general (Bierhoff, 2009). Beckman et al. (2008) show that permissions can also soundly verify typestates in concurrent programs that use Atomic blocks and in fact *enforce* the use of mutual exclusion primitives where thread-shared data is accessed (Beckman et al., 2008). Considering these recent developments, the author hopes and believes that we have only just begun to explore the possibilities of using permissions to improve our programs.

## 1.4 Thesis Outline

The remainder of this dissertation is organized into 8 chapters. Chapter 2 states the thesis of this dissertation and supporting hypotheses. Chapter 3 explains our approach in detail based on two familiar examples, iterators and streams. Both are inspired by APIs in the Java standard library. The iterator example shows how even complex protocols can be specified and gives an intuition for how to check compliance to protocols using access permissions. The stream example illustrates how we can verify API *implementation code*, including the thorny issue of how to deal with inheritance. This chapter also provides a brief introduction to the linear logic connectives we use.

Chapter 4 presents a formal account of an object-oriented language and type system based on the ideas of state refinement and access permissions. Protocol checking amounts to proving linear logic predicates using the conventional proof rules for that logic augmented with a judgment for permission splitting and merging. The soundness proof for a fragment of this type system appears in Bierhoff and Aldrich (2007a). It guarantees that declared protocols are never violated at run-time.

Chapter 5 presents an inference algorithm for tracking polymorphic permissions. While the type system

mentioned above "guesses" proofs in our augmented linear logic, the inference algorithm can *automatically* generate these proofs. This is a crucial step for enabling automated protocol checking.

Chapter 6 describes how this inference algorithm has been embedded into practical tooling for the Java language that is based on a branch-sensitive dataflow analysis. Java annotations can be used to describe protocols and make protocol checking a modular problem that can be tackled by analyzing each method in the program separately. The chapter discusses several extensions to the theory of access permissions developed in Chapters 4 and 5 to make the tool more useful in practice and shows how practical language features such as constructors can be handled.

Chapter 7 summarizes case studies in specifying major APIs from the Java standard library and checking open-source codebases with the tool described in Chapter 6. Specified APIs include the Java Collections framework and the JDBC library that was used as an example throughout this chapter. We checked compliance to these API protocols in a part of the Apache Beehive library and checked iterator-related protocols in PMD, an open-source bug-finding tool.

Finally, Chapter 8 puts this dissertation in the context of related work and Chapter 9 concludes with a discussion of the lessons to be learned from this research and future work it motivates.

**Typefaces.** We use the following formatting conventions:

- `Monospace` is used for code embedded into the text, such as an `if` statement.
- Sans serif is used for typestates such as `open`, although we occasionally use *italics* to refer to the conceptual notion of a state.
- CAPS is used for naming inference rules.

# Chapter 2

# Thesis and Hypotheses

## 2.1 Thesis

> *Typestate-based protocols with state refinement and access permissions can be used for automated, static, modular enforcement of API protocols in practical object-oriented software.*

This thesis sets the bar high in claiming the applicability of our approach to "practical software". What we have in mind with this charged word is to show that our approach can be used with common software practices—which in particular include the APIs and programming languages in practical use, as well as how code is commonly written—as opposed to only being applicable to toy programs. For this dissertation we approximate this standard as follows: we chose commonly used APIs from the Java standard library and released code from open-source software projects as study subjects in several case studies.

We do not believe that our approach would work well with every conceivable API or piece of code, and in particular not with all off-the-shelf—found in the wild—code *as–is*. But we will provide evidence that our approach is well-suited for expressing and enforcing challenging protocols for real APIs, and that it can achieve good precision when checking protocols in off-the-shelf code. Furthermore, small code changes can sometimes overcome reasoning imprecisions.

Our approach therefore has properties similar to a conventional type system such as the one found in Java: small code changes sometimes allow successful protocol checking, and runtime checks in the program can be used to overcome reasoning imprecisions, similar to typecasts in object-oriented type systems.

## 2.2 Hypotheses

In an ideal scenario, we would evaluate our approach by deploying it to software engineers in real-world software development projects, or by using it on a large number of software artifacts. But an evaluation on this scale is beyond the scope of this dissertation.

Instead we chose to focus on four testable hypotheses that address the predominant concerns of this research: they aim at establishing our approach's *well-foundedness* and its *applicability* to "practical software"—as explained above—in particular with respect to APIs, conventional programming languages, and how code is written in practice.

This section discusses each of these hypotheses in detail and previews the validation this dissertation provides for them. Our first hypothesis is that common protocols can be captured succinctly with our approach, establishing applicability to APIs. Our second hypothesis is that protocol compliance can be checked in a sound modular fashion, showing our approach's well-foundedness.[1] The third hypothesis is that the approach can be embodied in tools for conventional programming languages that impose annotation burden for developers that is comparable to type declarations, establishing the approach's applicability to commonly used languages. Finally, the fourth hypothesis is that our approach can be used to check protocols in off-the-shelf software, which shows the approach's applicability to code written in practice.

### 2.2.1 Capture Common Protocols Succinctly

Our approach requires relevant protocols to be specified before they can be checked. We were concerned whether interesting protocols could be specified in a concise way. Our first hypothesis therefore is:

> Typestate-based specifications with access permissions can succinctly capture API protocols commonly occurring in practice.

We support this hypothesis with the following evidence:

1. Case studies on several Java APIs in wide practical use (Section 7.1) showing that our approach can express commonly-used protocols.
2. Identification of a number of challenging recurring patterns, all of which can be expressed with our approach (Section 7.4.1).
3. Measurements showing that the number of annotations needed for capturing protocols with our approach is small, especially compared to informal documentation.

These results establish the applicability of our approach to API protocols occurring in practice.

### 2.2.2 Sound Modular Checking

In order to "statically enforce protocols" as stated in the thesis and therefore provide positive assurance at compile-time that protocols are followed we have to make sure that our approach is well-founded. Our second hypothesis expresses this concern:

> Typestate-based protocols with access permissions can be verified in a sound modular fashion.

This hypothesis is supported by a formalization of our approach as a linear dependent type system (Chapter 4) and a proof of soundness (Bierhoff and Aldrich, 2007a). The proof of soundness, informally speaking, guarantees that programs that typecheck in our type system will never violate declared protocols at run-time.

The formalization characterizes the theoretical properties of our approach, which include soundness, modularity, the approach's character as a refinement type system, its aliasing flexibility, and its support for dynamic state tests and flexible overriding in subclasses.

[1] Applicability to APIs precedes the well-foundedness hypothesis because without being able to capture common protocols there would be no point in checking them.

### 2.2.3 Automation

We were concerned to which extent programs could be checked automatically for protocol compliance. In particular, if the approach required manual proofs that programs are consistent with protocols then its usability by software developers would be greatly reduced. Our second hypothesis is therefore:

> Access permission-based protocol checking can be automated with annotation burden comparable to conventional type declarations and used with conventional programming languages.

We support this hypothesis as follows:

1. We describe a type inference system inspired by constraint logic programming that reduces tracking permissions through code to linear constraints (Chapter 5).
2. We embedded this inference algorithm into a tool, Plural, for checking protocols in the Java programming language (Chapter 6). The tool requires developers to provide annotations for method parameters and fields, but processes individual Java methods fully automatically.
3. Measurements showing that developer-provided annotations are in size comparable to conventional Java typing information (both for describing API protocols and for tracking objects from APIs in client code).
4. Measurements showing that Plural checks protocols on average fast enough for interactive use.

These results emphasize that our approach is amenable to automated reasoning, which makes it interesting for use in practice.

### 2.2.4 Practical Checking

As with any sound approach to a statically undecidable problem, there will be reasoning imprecisions which can lead to false positives when using our approach for checking protocols. We were concerned that large numbers of false positives would make the approach impractical to use. Consequently, our final hypothesis is the following:

> The approach can enforce API protocols in off-the-shelf object-oriented software.

We support this hypothesis with the following evidence:

1. Case studies showing that the tool can check protocols of Java APIs in existing open-source codebases with few false positives (Chapter 7).
2. A comparison between our approach and the leading existing approach in checking protocol compliance under aliasing showing that our approach successfully checks protocol compliance in cases where existing approaches cannot (Section 7.3.3).
3. A categorization of remaining false positives that shows that most false positives can be removed with simple improvements to the approach and tooling. Furthermore, refactorings sometimes enable automated protocol checking in code that could not be analyzed as written.

These results establish applicability of our approach to software found in practice. They do *not* mean that it will be straightforward to apply our approach to every conceivable piece of existing code, but they show that our approach is flexible enough to check protocols in code written to serve a practical purpose with reasonable precision.

# Chapter 3

# Approach: Access Permissions

This chapter introduces the major elements of the proposed approach with examples from the Java standard library. Section 3.1, on Java iterators, focuses on specifying protocols and checking client code. Stream implementations in the Java standard library are used to illustrate checking of API implementation code against declared protocols and our handling of subtyping and inheritance (Section 3.2).

This chapter neither attempts to evaluate the approach presented in this dissertation, nor does the order in which ideas are presented reflect the genesis of the approach. The chapter uses a series of convenient examples to informally introduce the major elements of the approach. Chapter 7 evaluates the approach in several case studies.

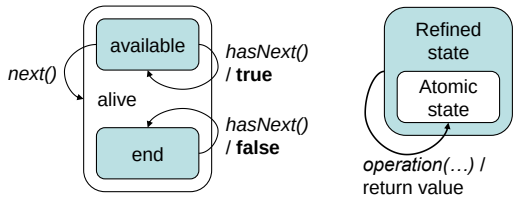
## 3.1 Java Iterators

This section illustrates specification of protocols and verification of client code based on a case study with Java *iterators* (Bierhoff, 2006). Iterators follow a straightforward protocol, but define complicated aliasing restrictions that are easily violated by developers. They are therefore a good vehicle to introduce our approach to handling aliasing in protocol verification.

### 3.1.1 Specification Goals

The specification presented in this section models the `Iterator` interface defined in the Java standard library. We will focus on *read-only* iterators, i.e., iterators that cannot modify the collection they iterate over. We will refer to read-only iterators simply as "iterators" and qualify full Java iterators as "modifying iterators". Full iterators will be discussed in Section 3.1.6. Goals of the presented specification include the following.

- Capture the usage protocol of Java iterators.
- Allow creating an arbitrary number of iterators over collections.
- Invalidate iterators before the iterated collection is modified.




(a) Protocol (b) Legend

Figure 3.1: Read-only iterator state machine protocol. Return values for `hasNext` are included to show the possible outcomes of dynamic state tests.

### 3.1.2 State Machine Protocol

An iterator returns all elements of an underlying *collection* one by one. Collections in the Java standard library are lists or sets of objects. Their interface includes methods to add objects, remove objects, and test whether an object is part of the collection (see Figure 3.3). The interface also defines a method `iterator` that creates a new iterator over the collection.

The `Iterator` interface defines two methods `hasNext` and `next`. Every call to `next` returns another object contained in the iterated collection. The method `hasNext` determines whether `next` can be called. It is *illegal* to call `next` once `hasNext` returns `false`. Figure 3.1 illustrates this protocol as a state machine.

The method `hasNext` determines whether another object is `available` or the iteration reached its `end`. We call `hasNext` a *dynamic state test*: its return value indicates what state the iterator is currently in. Notice that `hasNext` can always be called and does not change state. The next section will show how this protocol can be specified.

### 3.1.3 Iterator Interface Specification

**States through refinement.** Following previous work we call the set of possible states of an object its *state space* and define it as part of the object's interface. As suggested above, we can model the iterator state space with two states, `available` and `end`. In our approach, states are introduced by *refinement* of an existing state. State refinement corresponds to OR-states in Statecharts (Harel, 1987) and puts states into a hierarchy.

State refinement allows interfaces to, at the same time, *inherit* their supertypes' state spaces, define additional (more fine-grained) states, and be properly *substitutable* as subtypes of extended interfaces (Bierhoff and Aldrich, 2005). Refinement guarantees substitutability because all new states defined in a subtype correspond to a state inherited from the supertype. States form a hierarchy rooted in a state `alive` defined in the root type `Object`. For instance, the state space for iterators can be defined as follows.

```
states available, end refine alive;
```

Typestates do *not* correspond to fields in a class. They describe an object's state of execution abstractly and information about fields can be *tied* to typestates using state invariants (see Section 3.2.1).

**Access permissions capture design intent.** Iterators have only two methods, but these have very different behavior. While `next` can *change* the iterator's state, `hasNext` only *tests* the iterator's state. And even when a call to `next` does not change the iterator's typestate, it still advances the iterator to the next object in the sequence. `hasNext`, on the other hand, treats the iterator as *pure*: it does not modify the iterator at all.

We use a novel abstraction, *access permissions* ("permissions" for short), to capture this *design intent* as part of the iterator's protocol. Permissions are associated with object references and govern how objects can be accessed through a given reference (Boyland, 2003). For `next` and `hasNext` we need only two kinds of permissions; additional permissions will be introduced later.

- full permissions grant read/write access to the referenced object *and guarantee that no other reference has write access* to the same object.
- pure permissions grant read-only access to the referenced object but *assume that other permissions could modify* the object.

A distinguished full permission can co-exist with an arbitrary number of pure permissions to the same object. This property will be enforced when verifying protocol compliance. In a specification we write $\mathit{perm}(x)$ for a permission to an object referenced by $x$, where *perm* is one of the permission kinds. Access permissions carry state information about the referenced object. For example, "$\mathsf{full}(\mathit{this})$ **in** available" represents a full permission for the receiver object, which we refer to with *this*, that is in the available state.

**Linear logic specifications relate objects.** Methods can be specified with a *state transition* that describes how method parameters change state during method execution. Access permissions represent resources that have to be consumed upon usage—otherwise permissions could be freely duplicated, possibly violating other permissions' assumptions. In particular, permissions used in a method call have to be taken away from the caller and replaced by whatever permissions the method returns.

Therefore, we base protocol specifications on linear logic (Girard, 1987). Linear logic treats facts as resources that are consumed when used to prove other facts. This matches our intuition of permissions as resources. Methods are specified with a linear implication ($\multimap$) from pre- to post-condition. Most pre- and post-conditions shown in this chapter use conjunction ($\otimes$) and disjunction ($\oplus$) to relate permissions.[1] In certain cases, internal choice ($\&$, also called additive conjunction) has been useful (Bierhoff, 2006). These connectives represent the decidable multiplicative-additive fragment of linear logic (MALL).

The `next` method illustrates that state transitions can be non-deterministic: the method makes an "internal choice", meaning that a client cannot predict whether the iterator will be in the available or the end state when `next` returns. We can express this uncertainty by using alive in `next`'s post-condition. In a Statechart, this corresponds to transitioning to an OR-state (Figure 3.1). Furthermore, `next` modifies the iterator (by advancing it to the next element in the collection. Thus, we can specify `next` so that it requires a full permission in state available and returns the full permission in the alive state, which we write as follows:

$$\mathsf{full}(\mathit{this})\ \textbf{in}\ \mathsf{available} \multimap \mathsf{full}(\mathit{this})\ \textbf{in}\ \mathsf{alive}$$

Dynamic state tests (like `hasNext`) require relating the (Boolean) method result to the state of the tested object (usually the receiver). A disjunction of conjunctions expresses the two possible outcomes of `hasNext`

[1] "Tensor" ($\otimes$) corresponds to conjunction, "alternative" ($\oplus$) to disjunction, and "lolli" ($\multimap$) to implication in conventional Boolean logic. The key difference is that linear logic treats known facts as resources that are consumed when proving another fact.

where each conjunction relates a possible method result to the corresponding state of the receiver object:

$$\begin{aligned}\mathsf{pure}(\mathit{this}) \multimap (\mathit{result} = \texttt{true} \otimes \mathsf{pure}(\mathit{this}) \ \textbf{in}\ \mathsf{available}) \\ \oplus\ (\mathit{result} = \texttt{false} \otimes \mathsf{pure}(\mathit{this}) \ \textbf{in}\ \mathsf{end})\end{aligned}$$

Note that we refer to the method return value with *result*. We use the convention that $\multimap$ binds weaker than $\otimes$ and $\oplus$. Finally, we will often omit the default state information " **in** alive", as in the above precondition.

These specifications enforce the characteristic alternation of calls to `hasNext` and `next`: `hasNext` determines the iterator's current state. If it returns `true` then it is legal to call `next`. The iterator is in an unknown state after `next` returns, and another `hasNext` call determines the iterator's new state. Clients must call `hasNext` at least once before each call to `next` in order to establish the available state required for calling `next`.

### 3.1.4 Creating and Disposing Iterators

Multiple (independent) iterators are permitted for a single collection at the same time. However, the collection must not be modified while iteration is in progress. Standard implementations try to detect violations of this rule, so-called *concurrent modifications*, on a best-effort basis. But, ultimately, Java programmers have to make sure on their own that collections are not modified while iterated. Following Ramalingam et al. (2002) we note that "concurrent" modifications often occur in single-threaded programs, for instance when new elements are added to a list during an iteration of the existing items in the list.

This section shows how concurrent modifications can be prevented by establishing aliasing constraints between iterators and their collections. As we will see, this problem is largely orthogonal to specifying the relatively simple protocol for individual iterators that was discussed in the previous section.

**Immutable access prevents concurrent modification.** Access permissions can guarantee the absence of concurrent modification (as explained above). The key observation is that when an iterator is created it stores a reference to the iterated collection in one of its fields. This reference should be associated with a permission that guarantees the collection's *immutability* while iteration is in progress. We include two previously proposed permissions (Boyland, 2003) into our system in order to properly specify collections.

- immutable permissions grant read-only access to the referenced object *and guarantee that* no *reference has read/write access* to the same object.
- unique permissions grant read/write access *and guarantee that no other reference has* any *access* to the object.

Thus immutable permissions cannot co-exist with full permissions to the same object. We can specify the collection's `iterator` method using these permissions as follows. Notice how it *consumes* or *captures* the incoming receiver permission and returns an initial unique permission to a fresh iterator object.

```
public class Collection {
  Iterator iterator() : immutable(this) ⊸ unique(result)
}
```

```
Collection c = new ...;
Iterator it = c.iterator(); // legal
while(it.hasNext() && ...) { // legal
    Object o = it.next(); // legal
    Iterator it2 = c.iterator(); // legal
    while(it2.hasNext()) { // legal
        Object o2 = it2.next(); // legal
        /*...*/ }
}
if(it.hasNext() && c.size() == 3) { // legal
    c.remove(it.next()); // legal
    if(it.hasNext()) /*...*/ } // ILLEGAL: Iterator access after iterated collection was modified
Iterator it3 = c.iterator(); // legal
```

Figure 3.2: Simple `Iterator` client with concurrent modification error

It turns out that this specification precisely captures Sun's Java standard library implementation of iterators: iterators are realized as inner classes that implicitly reference the collection they iterate. Using an immutable permission for this reference prevents concurrent modifications (see below). If other references might modify the collection then this specification does not allow iterators to be created.

**Permission splitting.** How can we track permissions? Consider a client such as the one in Figure 3.2. It gets a unique permission when first creating a collection. Then it creates an iterator that captures an immutable permission to the collection. However, the client later needs more immutable permissions to create additional iterators. Thus while a unique permission is intuitively stronger than an immutable permission we cannot just coerce the client's unique permission to an immutable permission and pass it to `iterator`: it would get captured by the newly created iterator, leaving the client with no permission to the collection at all.

In order to avoid this problem we use *permission splitting* in our verification approach. Before method calls we split the original permission into two, one of which is retained by the caller. Permissions are split so that their assumptions are not violated. In particular, we never duplicate a full or unique permission and make sure that no full permission co-exists with an immutable permission to the same object. Some of the legal splits are the following.

$$\begin{aligned} \mathsf{unique}(x) &\Rrightarrow \mathsf{full}(x) \otimes \mathsf{pure}(x) \\ \mathsf{full}(x) &\Rrightarrow \mathsf{full}(x) \otimes \mathsf{pure}(x) \\ \mathsf{full}(x) &\Rrightarrow \mathsf{immutable}(x) \otimes \mathsf{immutable}(x) \\ \mathsf{immutable}(x) &\Rrightarrow \mathsf{immutable}(x) \otimes \mathsf{immutable}(x) \end{aligned}$$

They allow the example client in Figure 3.2 to retain an immutable permission when creating iterators, permitting multiple iterators and reading the collection directly at the same time.

**Permission merging recovers modifying access.** When splitting a full permission to a collection into immutable permissions we lose the ability to modify the collection. Intuitively, we would like to reverse permission splits to regain the ability to modify the collection.

```
interface Iterator<c : Collection, k : Fract> {
  states available, end refine alive

  boolean hasNext() :
      pure(this) ⊸ (result = true ⊗ pure(this) in available)
                 ⊕ (result = false ⊗ pure(this) in end)
  Object next() :
      full(this) in available ⊸ full(this)
  void finalize() :
      unique(this) ⊸ immutable(c, k)
}

interface Collection {
  void add(Object o) : full(this) ⊸ full(this)
  int size() : pure(this) ⊸ result ≥ 0 ⊗ pure(this)
  // remove(), contains() etc.  similar

  Iterator<this, k> iterator() :
      immutable(this, k) ⊸ unique(result)
}
```

Figure 3.3: Read-only `Iterator` and partial `Collection` interface specification

Such *permission merging* can be allowed if we introduce the notion of fractions (Boyland, 2003). Essentially, fractions keep track of how often a permission was split. This later allows the merging of permissions (with known fractions) by adding together their fractions. A unique permission by definition holds a *full fraction* that is represented by one (1); the exact values of other fractions do not carry any semantic significance. We will capture fractions as part of our permissions and write $perm(x, k)$ for a given permission with fraction $k$. We usually do not care about the exact fraction and therefore implicitly quantify over all fractions. If a fraction does not change we often will omit it. Fractions allow us to define splitting and merging rules such as the following:

$$\begin{aligned}
\mathsf{unique}(x, 1) &\iff \mathsf{full}(x, 1/2) \otimes \mathsf{pure}(x, 1/2) \\
\mathsf{full}(x, k) &\iff \mathsf{full}(x, k/2) \otimes \mathsf{pure}(x, k/2) \\
\mathsf{full}(x, 1/2) &\iff \mathsf{immutable}(x, 1/4) \otimes \mathsf{immutable}(x, 1/4) \\
\mathsf{immutable}(x, k) &\iff \mathsf{immutable}(x, k/2) \otimes \mathsf{immutable}(x, k/2)
\end{aligned}$$

For example, we can split $\mathsf{full}(it, 1/2)$ into $\mathsf{full}(it, 1/4) \otimes \mathsf{pure}(it, 1/4)$ and recombine them. Such reasoning lets our iterator client recover a unique iterator permission after each call into the iterator.

**Recovering collection permissions.** Iterators are created by trading a collection permission for a unique iterator permission. We essentially allow the opposite trade as well in order to modify a previously iterated collection again: We can safely consume a unique iterator permission and recover the permissions to its fields because no reference will be able to access the iterator anymore. We could do this at any time, but a good time to do so is when an iterator is no longer used. A simple live variable analysis can identify

```
Collection c = new ...                                          unique(c)
Iterator it<c, 1/2> = c.iterator();                             immutable(c, 1/2) ⊗ unique(it)
while(it.hasNext() && ...)  {                                   immutable(c, 1/2) ⊗ unique(it) in available
    Object o = it.next();                                       immutable(c, 1/2) ⊗ unique(it)
    Iterator it2<c, 1/4> = c.iterator();                        immutable(c, 1/4) ⊗ unique(it) ⊗ unique(it2)
    while(it2.hasNext()) {                                      immutable(c, 1/4) ⊗ unique(it) ⊗ unique(it2) in available
        Object o2 = it2.next();                                 immutable(c, 1/4) ⊗ unique(it) ⊗ unique(it2)
        ...  }                                                  // it2 dies
}                                                               immutable(c, 1/2) ⊗ unique(it)
if(it.hasNext() && c.size() == 3) {                             immutable(c, 1/2) ⊗ unique(it) in available
    c.remove(it.next()); // kill iterator "it" after next()     unique(c) and no permission for "it"
    if(it.hasNext()) ...  }                                     // ILLEGAL
// it definitely dead                                           unique(c)
Iterator it3<c, 1/2> = c.iterator();                            immutable(c, 1/2) ⊗ unique(it3)
```

Figure 3.4: Verifying a simple `Iterator` client

when variables with `unique` permissions are no longer used. (As a side effect, a permission-based approach therefore allows identifying dead objects.)

For lack of a more suitable location, we annotate the `finalize` method to indicate what happens when an iterator is no longer usable. And in order to re-establish *exactly* the permission that was originally passed to the iterator we parameterize `Iterator` objects with the collection instance they iterate ($c$) and the collection permission's fraction ($k$). The `finalize` specification can then release the captured collection permission from dead iterators. Figure 3.3 summarizes the complete specification for iterators and a partial collection specification. The specifications augment the conventional Java interfaces without changing their type signatures.

### 3.1.5 Client Verification

Figure 3.4 illustrates how our client from Figure 3.2 can be verified by tracking permissions and splitting/merging them as necessary. After each line of code we show the current set of permissions on the right-hand side of the figure. We recover collection permissions by "killing" dead iterators as soon as possible (applying their specification for `finalize`). We correctly identify the seeded protocol violation.

### 3.1.6 Modifying Iterators

Actual Java iterators can modify the iterated collection: the additional method `remove` removes the most recently retrieved element from the underlying collection. This invalidates all *other* iterators over the same collection, but *not* the iterator at hand.

**State dimensions for orthogonal concerns.** We propose *state dimensions* to address separate concerns with states that are independent from each other (Bierhoff and Aldrich, 2005). State dimensions are orthogonal sets of mutually exclusive states and correspond to AND-states in Statecharts (Harel, 1987).

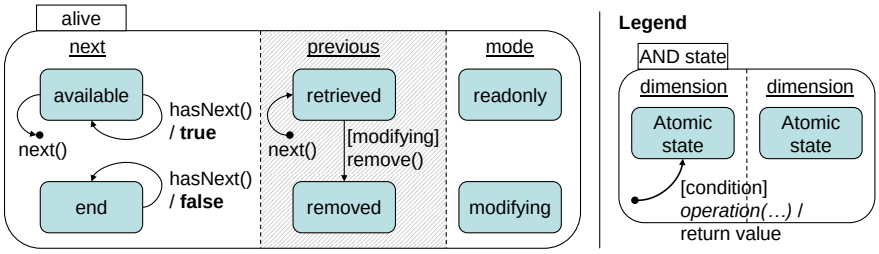



Figure 3.5: Modifying iterator state machine protocol. The shaded area is affected by `remove`.

We model full Java iterators with three orthogonal dimensions (Figure 3.5). At runtime, an iterator object will be in exactly one of the states from each dimension. Notice that we introduce dimensions by simply refining the same state more than once (Figure 3.6). The three iterator dimensions have the following meaning (Figure 3.5):

- `next` captures whether another element is available or not. This corresponds to the state space of read-only iterators.
- `previous` captures whether an element can be removed or not. Notice that removing an element is only possible if an element was retrieved but not removed yet.
- `mode` indicates whether the current iterator is read-only or modifying. Once created, iterators do not change state in this dimension.

These dimensions encode 8 unique state combinations (there are 2 states each in 3 dimensions). In the following we show how dimensions can be used for concise protocol specifications and aliasing control.

**Permissions for dimensions.** Iterator methods only access certain state dimensions: `hasNext` only looks at the `next` dimension, `remove` only modifies the `previous` dimension, and `next` modifies `next` and `previous`. We include this information as an additional parameter into access permissions and, e.g., write full(*this*, previous) for a permission for the `previous` dimension. It corresponds to an "area" in the iterator Statechart within which the permission can change state. For example, the shaded area in Figure 3.5 corresponds to the permission needed to call `remove`. Notice that separate permissions can exist for orthogonal dimensions.

Figure 3.6 shows a specification for full Java iterators. Notice how the different methods use permissions for different state dimensions. In Section 7.1.2 we will see how to specify the `Collection`'s `iterator` method so that it leaves the decision whether an iterator is modifying or read-only up to the client.

## 3.2 Java Stream Implementations

I/O protocols are common examples for typestate-based protocol enforcement approaches (DeLine and Fähndrich, 2001, 2004b; Bierhoff and Aldrich, 2005). This section summarizes a case study in applying our

```
interface Iterator<c : Iterable, k : Fract> {
  next = available, end refine alive
  previous = retrieved, removed refine alive
  mode = readonly, modifying refine alive

  boolean hasNext() : pure(this, next) ⊸
       (result = true ⊗ pure(this, next) in available)
       ⊕ (result = false ⊗ pure(this, next) in end)
  Object next() :
    full(this, next) in available ⊗ full(this, previous) ⊸
      full(this, next) ⊗ full(this, previous) in retrieved
  void remove() :
    full(this, previous) in retrieved ⊗ immutable(this, modifying) ⊸
      full(this, previous) in removed ⊗ immutable(this, modifying)
  void finalize() :
     (unique(this) in readonly ⊸ immutable(c, k))
    & (unique(this) in modifying ⊸ full(c, k))
}
```

Figure 3.6: Java modifying `Iterator` specification

approach to Java *character streams* and in particular *stream pipes* and *buffered input streams*. The section focuses on *implementation verification* of stream classes, which—to our knowledge—has not been attempted with typestates before. Implementation verification generalizes techniques shown in the previous section for client verification. This dissertation assumes that concurrent object accesses are correctly synchronized. A formal treatment of access permissions in multi-threaded programs can be found in Beckman et al. (2008).

### 3.2.1 Stream Pipes

Pipes carry alphanumeric *characters* from a source to a sink. Pipes are commonly used in operating system shells to forward output from one process to another process. The Java I/O library includes a pair of classes, `PipedOutputStream` and `PipedInputStream`, that offers this functionality inside Java applications. This section provides a specification for Java pipes and shows how the classes implementing pipes in the Java standard library can be checked using our approach.

**Informal pipe contract.** In a nutshell, Java pipes work as follows: a character-producing "writer" `writes` characters into a `PipedOutputStream` (the "source") that forwards them to a connected `PipedInputStream` (the "sink") from which a "reader" can `read` them. The source forwards characters to the sink using the internal method `receive`. The writer calls `close` on the source when it is done, causing the source to call `receivedLast` on the sink (Figure 3.8).

The sink caches received characters in a circular buffer. Calling `read` on the sink removes a character from the buffer (Figure 3.9). Eventually the sink will indicate, using an *end of file token* (EOF, -1 in Java), that no more characters can be read. At this point the reader can safely close the sink. Closing the sink before EOF was read is unsafe because the source with throw an exception when the writer writes more characters to the pipe.

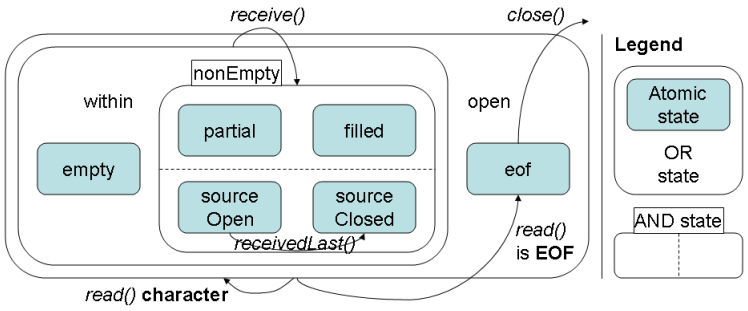



Figure 3.7: `PipedInputStream`'s state space (inside `open`)

The pipe classes in Sun's standard library implementation have built-in runtime checks that throw exceptions in the following error cases: (1) closing the sink before the source, (2) writing to a closed source or pushing characters to the sink after the source was closed, and (3) reading from a closed sink. The specification we present here makes these error cases impossible.

**State space with dimensions.** The *source protocol* can be modeled with three states `raw`, `open`, and `closed`. `raw` indicates that the source is not connected to a sink yet. For technical reasons that are discussed below, we refine `open` into `ready` and `sending`. The writer will always find the source in state `ready`.

For the *sink protocol* we again distinguish `open` and `closed`. A refinement of `open` helps capture `read`'s protocol: The sink is in the `within` state as long as `read` returns characters; the `eof` state has been reached once `read` returns the EOF token. While `within`, we keep track of the sink's buffer being `empty` or `nonEmpty`. We further refine `nonEmpty` into `partial` and `filled`, the latter corresponding to a full buffer.

At the same time, however, we would like to keep track of whether the source was closed, i.e., whether `receivedLast` was called. To capture this, we refine `nonEmpty` twice, along different dimensions. We call the states for the second dimension `sourceOpen` and `sourceClosed` with the obvious semantics. Note that we only need the additional `source` dimension while the buffer is `nonEmpty`; the source is by definition open (closed) in the `empty` (`eof`) state.[2] To better visualize the sink's state space, Figure 3.7 summarizes it as a Statechart.

**Shared modifying access.** Protocols for source and sink are formalized in Figures 3.8 and 3.9 with specifications that work similar to the iterator example in the last section. However, the sink is conceptually modified through two distinct references, one held by the source and one held by the reader. In order to capture this, we introduce our last permission.

- `share` permissions grant read/write access to the referenced object and *allow other permissions to have read/write access as well.*

Conventional programming languages can be thought of as always using `share` permissions. Interestingly, `share` permissions are split and merged exactly like `immutable` permissions. Since `share` and im-

[2]This is only *one* way of specifying the sink. It has the advantage that readers need not concern themselves with the internal communication between source and sink.

mutable permissions cannot co-exist, our rules force a commitment to either one when initially splitting a full permission. Examples include the following:

$$\begin{array}{rcl} \mathsf{full}(x, 1/2) & \Leftrightarrows & \mathsf{share}(x, 1/4) \otimes \mathsf{share}(x, 1/4) \\ \mathsf{share}(x, k) & \Leftrightarrows & \mathsf{share}(x, k/2) \otimes \mathsf{share}(x, k/2) \\ \mathsf{share}(x, k) & \Leftrightarrows & \mathsf{share}(x, k/2) \otimes \mathsf{pure}(x, k/2) \end{array}$$

**State guarantees.** We notice that most modifying methods cannot change a stream's state arbitrarily. For example, `read` and `receive` will never leave the open state.

To formalize this observation, we can use the idea of confining permissions to a part of the state space that we introduced in Section 3.1.6: we can specify a *state guarantee*, i.e., a state that cannot be left with any permission. As before, a state guarantee (also called the permission's *root state*) corresponds to an "area" in a Statechart that cannot be left. As such, state guarantees are the result of combining permissions with hierarchical state refinement. They constrain the behavior of a method to a certain region in the state space. As an example, we can write the permission needed for `read` as $\mathsf{share}(\mathit{this}, \mathsf{open})$. We will usually omit the trivial state guarantee alive so that for instance $\mathsf{share}(\mathit{this}, \mathsf{alive})$ is the same as $\mathsf{share}(\mathit{this})$.

State guarantees turn out to be crucial in making share and pure permissions useful because they guarantee the referenced object to remain in the guaranteed state even after possible modifications through other permissions.

**Explicit fractions for temporary heap sharing.** When specifying the sink methods used by the source (`receive` and `receivedLast`) we have to ensure that the source can no longer call the sink after `receivedLast` so the sink can be safely closed. Moreover, in order to close the sink, we need to restore a permission rooted in alive. Thus the two share permissions for the sink have to be joined in such a way that there are definitely no other permissions relying on open (such permissions, e.g., could have been split off of one of the share permissions).

We again rely on fractions to accomplish this task. We use fractions to track, *for each state separately*, how many permissions rely on it. What we get is a *fraction function* that maps guaranteed states (i.e., the permission's root and its super-states) to fractions. For example, if we split an initial unique permission for a `PipedInputStream` into two share permissions guaranteeing open then these permissions rely on open and alive with a $1/2$ fraction each.[3]

In order to close the sink, we have to make sure that there are *exactly* two share permissions relying on open. Fraction functions make this requirement precise. For reasons of readability, we use the abbreviation half in Figures 3.8 and 3.9 that stands for the following permission.

$$\mathsf{half}(x, \mathsf{open}) \equiv \mathsf{share}(x, \mathsf{open}, \{\mathsf{alive} \mapsto 1/2, \mathsf{open} \mapsto 1/2\})$$

By adding fractions and moving the state guarantee up in the state hierarchy, the initial permission for the sink, $\mathsf{unique}(\mathit{this}, \mathsf{alive}, \{\mathsf{alive} \mapsto 1\})$, can be regained from two $\mathsf{half}(\mathit{this}, \mathsf{open})$ permissions.[4] half is the only permission with an explicit fraction function. All other specifications implicitly quantify over all fraction functions and leave them unchanged.

[3] Iterator permissions are rooted in alive and their fraction functions map alive to the given fraction.

[4] Any other combination of fractions, such as $1/4$ and $3/4$, will work as well.

```
public class PipedOutputStream {
  states raw, open, closed refine alive;
  states ready, sending refine open;

  raw := sink = null
  ready := half(sink, open)
  sending := sink ≠ null
  closed := sink ≠ null

  private PipedInputStream sink;

  public PipedOutputStream() : 1 ⊸ unique(this) in raw
  { }

  void connect(PipedInputStream snk) : full(this) in raw ⊗ half(snk, open) ⊸ full(this) in ready
  { sink = snk;                                                          store permission in field
  }                                                                      full(this) in ready

  public void write(int b) : full(this, open) in ready ⊗ b ≥ 0 ⊸ full(this, open) in ready
  {                                                                      half(sink, open) from invariant
    sink.receive(b);                                                     returns half(sink, open)
  }                                                                      full(this, open) in ready

  public void close() : full(this) in ready ⊸ full(this) in closed
  {                                                                      half(sink, open) from invariant
    sink.receivedLast();                                                 consumes half(sink, open)
  }                                                                      full(this) in closed
}
```

Figure 3.8: Java `PipedOutputStream` (simplified)

Existing modular typestate checkers (DeLine and Fähndrich, 2001, 2004b) cannot handle the pipe example because objects cannot change state once they are shared. Explicit fractions are unsatisfying, and we never used them in our case studies (Chapter 7). But the pipe example suggests that they can give crucial expressiveness.

**State invariants map typestates to fields.** We now have a sufficient specification for both sides of the pipe. In order to verify their implementations we need to know what typestates correspond to in implementations. Our implementation verification extends Fugue's approach of using *state invariants* to map states to predicates that describe the fields of an object in a given state (DeLine and Fähndrich, 2004b). In addition to invariants for atomic states, we allow state invariants for dimensions and refined states in order to capture invariants common to all substates of a given state or dimension (Bierhoff and Aldrich, 2005). In particular, conventional class invariants (Leavens et al., 1999) simply appear as state invariants for alive.

Figure 3.8 shows that the source's state invariants describe its three states in the obvious way based on the field `sink`. Notice that the invariant for ready uses the permission half(*sink*, open) to control access

```
class PipedInputStream {
  stream = open, closed refines alive;
  position = within, eof refines open;
  buffer = empty, nonEmpty refines within;
  filling = partial, filled refines nonEmpty;
  source = sourceOpen, sourceClosed refines nonEmpty;

  empty := in ≤ 0 ⊗ closedByWriter = false
  partial := in ≥ 0 ⊗ in ≠ out
  filled := in = out
  sourceOpen := closedByWriter = false
  sourceClosed := closedByWriter ⊗ half(this, open)
  eof := in ≤ 0 ⊗ closedByWriter ⊗ half(this, open)
  open := closedByReader = false
  closed := closedByReader = true

  private boolean closedByWriter = false;
  private volatile boolean closedByReader = false;
  private byte buffer[] = new byte[1024];
  private int in = -1, out = 0;

  public PipedInputStream(PipedOutputStream src) :
    full(src) in raw ⊸ half(this, open) ⊗ full(src) in open
  {                                              unique(this) in open ⇛ half(this, open) ⊗ half(this, open)
    src.connect(this);                                                  consumes one half(this, open)
  }                                                                     half(this, open) ⊗ full(src) in open

  synchronized void receive(int b) : half(this, open) ⊗ b ≥ 0 ⊸ half(this, open) in nonEmpty
  { while(in == out) ...  // wait a second                              half(this, open) in filled
                                                                        half(this, open) in empty ⊕ partial
    if(in < 0) { in = 0; out = 0; }
    buffer[in++] = (byte)(b & 0xFF);
    if(in >= buffer.length) in = 0; }                                   half(this, open) in nonEmpty

  synchronized void receivedLast() : half(this, open) ⊸ 1
  { closedByWriter = true; }                                            this is now sourceClosed or eof

  public synchronized int read() : share(this, open) ⊸
    (result ≥ 0 ⊗ share(this, open)) ⊕ (result = −1 ⊗ share(this, open) in eof)
  { ...  } // analogous to receive()

  public synchronized void close() : half(this, open) in eof ⊸ unique(this) in closed
  {                                              half(this, open) from eof invariant ⇛ unique(this, open)
    closedByReader = true;
    in = -1; }
}
```

Figure 3.9: Java `PipedInputStream` (simplified)

through the *sink* field just as through local variables.

The sink's state invariants are much more involved (Figure 3.9) and define, e.g., what the difference between an empty ($in < 0$) and a filled circular buffer ($in = out$) is. Interestingly, these invariants are all meticulously documented in the original Java standard library implementation for `PipedInputStream` (Bierhoff and Aldrich, 2005). The half permission to itself that the sink temporarily holds for the time between calls to `receivedLast` and `close` can be merged with the half permission required by `close` into a unique permission, which allows dropping the open state guarantee and verifying that `close` is indeed allowed to close the sink.

**Verification with invariants.** Implementation checking assumes state invariants implied by incoming permissions and tracks changes to fields. Objects *have to be in a state whenever they yield control to another object*, which means during method calls and returns. This means that the source has to transition to a state, called sending in Figure 3.8, before calling the sink. Such *intermediate* or *hidden states* help us deal with reentrant calls (details in Section 4.2.2). We call sending an intermediary state because the writer always finds the source ready and never in the sending state.

Figures 3.8 and 3.9 show how implementation checking proceeds for most of the source's and sink's methods. Note that **1** denotes a trivially true predicate. We show in detail how field assignments change the sink's state. The sink's state information is frequently a disjunction of possible states. Dynamic tests essentially rule out states based on contradictory predicates (that result in `false`). *All* of these tests are present in the original Java implementation; we removed additional non-null and state tests that are obviated by our approach. This not only shows how our approach forces necessary state tests but also suggests that our specifications could be used to *generate* such tests automatically, which we leave to future work.

### 3.2.2 Buffered Input Streams

A `BufferedInputStream` ("buffer") wraps another "underlying" stream and provides buffering of characters for more efficient retrieval. We will use this example to illustrate our approach to handling inheritance. Compared to the original implementation, we made fields "private" in order to illustrate calls to overridden methods using `super`. We omit intermediate states in this specification for brevity.

**Class hierarchy.** `BufferedInputStream` is a subclass of `FilterInputStream`, which in turn is a subclass of `InputStream`. `InputStream` is the abstract base class of all input streams and defines their protocol with informal documentation that we formalize in Figure 3.10. It implements *convenience methods* such as `read(int[])` in terms of other—abstract—methods. `FilterInputStream` holds an underlying stream in a field `s` and simply forwards all calls to that stream (Figure 3.10). `BufferedInputStream` overrides these methods to implement buffering.

**Frames.** The buffer occasionally calls overridden methods to read from the underlying stream. How can we reason about these internal calls? Our approach is based on Fugue's *frames* for reasoning about inheritance (DeLine and Fähndrich, 2004b). Objects are divided into frames, one for each class in the object's class hierarchy. A frame holds the fields defined in the corresponding class. We call the frame corresponding to the object's runtime type the *virtual frame*, referred to with normal references (such as *x*, *y*, and *this*). Inside a method, we refer to the current frame—corresponding to the class that the method

```
public abstract class InputStream {
  states open, closed refine alive;
  states within, eof refine open;

  public abstract int read() :
    share(this_fr, open) ⊸ (result ≥ 0 ⊗ share(this_fr, open))
                ⊕ (result = −1 ⊗ share(this_fr, open) in eof)
  public abstract void close() :
    full(this_fr, alive) in open ⊸ full(this_fr, alive) in closed

  public int read(byte[] buf) :
    share(this, open) ⊗ buf ≠ null ⊸ (result ≥ 0 ⊗ share(this, open))
                              ⊕ (result = −1 ⊗ share(this, open) in eof)
  { ...  for(...)
      ...  int c = this.read() ...  }
}

public class FilterInputStream extends InputStream {
  within := unique(s) in within
  eof := unique(s) in eof
  closed := unique(s) in closed

  private volatile InputStream s;

  protected FilerInputStream(InputStream s) :
    unique(s, alive) in open ⊸ unique(this_fr, alive) in open
  { this.s = s; }
  ...  // read() and close() forward to s
}
```

Figure 3.10: Java `FilterInputStream` forwards all calls to underlying `InputStream` (simplified)

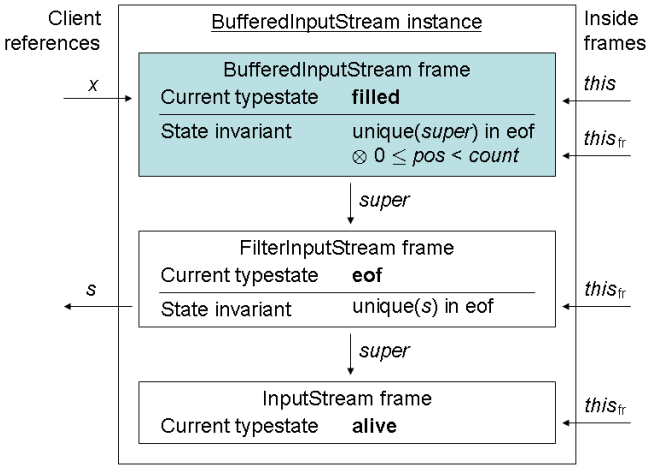



Figure 3.11: Frames of a `BufferedInputStream` instance in state *filled*. The shaded *virtual frame* is in a different state than its super-frame.

is defined in—with $this_{\text{fr}}$, and the frame corresponding to the immediate superclass is called *super* frame. Figure 3.11 shows a sample `BufferedInputStream` instance with its three frames.

**Frame permissions.** We interpret permissions to grant access to a particular frame. The permissions we have seen so far give a client access to the referenced object's virtual frame. Permissions for other frames are only accessible from inside a subclass through *super*.

Figure 3.11 illustrates that a `BufferedInputStream`'s state can differ from the state its `FilterInputStream` ("filter") frame is in: the filter's state might be `eof` (when the underlying stream reaches `eof`) while the buffer's is still `within` (because the buffer array still holds unread characters). The state invariants in Figure 3.12 formalize this. They let us verify that `super` calls in the buffer implementation respect the filter's protocol. Notice that the buffer methods call their overridden filter methods only *sometimes* (Figure 3.12). Our approach of decoupling states of different frames lets us verify this "selective" calling pattern.

Because the states of frames can differ it is important to enforce that a permission is only ever used to access fields in the frame it grants permission to. In specifications we specifically mark permissions that will actually access fields (and not just call other methods) of the receiver with $this_{\text{fr}}$. We require all methods that use these permissions to be overridden.[5] On the other hand, convenience methods such as `read(int[])` can operate with permissions to the virtual frame and need not be overridden (Figure 3.10).

This distinction implies that `fill` (Figure 3.12) *cannot* call `read(int[])` (because it does not have a suitable virtual frame permission) but *only* `super.read()`. This is imperative for the correctness of `fill` because a dynamically dispatched call would lead back into the—still `empty`—buffer, causing an infinite loop. (One can trigger exactly this effect in the Java 6 implementation of `BufferedInputStream`.)

[5] In the trivial case, the overriding method would simply call the overridden method with a *super*-call.

```
public class BufferedInputStream extends FilterInputStream {
  states depleted, filled refine within;

  closed := unique(super) in closed ⊗ buf = null
  open := unique(buf) ⊗ buf.length > 0 ⊗ count ≤ buf.length ⊗ pos ≥ 0
  filled := pos < count ⊗ unique(super) in open
  depleted := pos = count ⊗ unique(super) in within
  eof := pos = count ⊗ unique(super) in eof

  private byte buf[] = new byte[8192];
  private int count = 0, pos = 0;

  public BufferedInputStream(InputStream s) :
    unique(s) in open —∘ unique(this_fr) in open
  {                                                    count = pos = 0 ⊗ unique(buf)
    super(s);                                          unique(super) in open
  }                                                    unique(this_fr, alive) in open

  public synchronized int read() {
    if(pos >= count)
    {                                                  share(this_fr, open) in depleted ⊕ eof
      fill();                                          share(this_fr, open) in filled ⊕ eof
      if(pos >= count)
        return -1;                                     returns share(this_fr, open) in eof
    }                                                  any path: share(this_fr, open) in filled
    return buf[pos++] & 0xFF;
  }                                                    share(this_fr, open) in filled ⊕ eof

  private void fill() :
    share(this_fr, open) in depleted ⊕ eof —∘ share(this_fr, open) in filled ⊕ eof
  {                                                    invariant: unique(super) in within ⊕ eof
    count = pos = 0;                                   note: assumes buffer was fully read
    int b = super.read();                              unique(super) in within ⊕ eof
    while(b >= 0) {                                    unique(super) in within
      buf[count++] = (byte) (b & 0xFF);                share(this_fr, open) in filled
      if(count >= buf.length) break;
      b = super.read();                                unique(super) in within ⊕ eof
    }                                                  if loop never taken, share(this_fr, open) in eof
  }                                                    share(this, open) in filled ⊕ eof

  public synchronized void close() {
    buf = null;                                        invariant: unique(super) in open
    super.close();                                     unique(super) in closed
  }                                                    full(this_fr, alive) in closed
```

Figure 3.12: `BufferedInputStream` caches characters from `FilterInputStream` base class

## 3.3 Summary

This chapter first presented a specification of Java iterators. Our approach is to associate collections and iterators with access permissions, define a simple state machine to capture the iterator usage protocol, and track permission information using a decidable fragment of linear logic. Our logic-based specifications can relate objects to precisely specify method behavior in terms of typestates and support reasoning about dynamic tests.

This chapter also showed that our approach can verify that realistic Java pipe and buffered input stream implementations implement the declared protocol. Overall, we introduced five different permissions (Table 1.1). While three are adapted from existing work (Boyland, 2003; DeLine and Fähndrich, 2004b), we proposed full and pure permissions (Bierhoff, 2006). Permission splitting and joining is flexible enough to model temporary aliasing on the stack (during method calls) and in the heap (e.g., in pipes and iterators). We handle inheritance based on frames and permit dynamic dispatch within objects for convenience methods.

In this chapter we chose to make the linear logic formalism underlying our approach explicit. Our tool provides annotations that shield users from the complexities of linear logic in most all cases (Chapter 6).

The following chapter will treat the approach formally and chapters 5 and 6 show how reasoning with the approach can be automated in a tool.

# Chapter 4

# Type System

This chapter presents a formal treatment of state refinements and access permissions as introduced in the previous chapter in a linear dependent type system. The proof of soundness for a fragment of the presented calculus appears as a technical report (Bierhoff and Aldrich, 2007a) and is summarized in section 4.2.6. A previous version of this chapter appeared at the OOPSLA 2007 conference (Bierhoff and Aldrich, 2007b, sections 4 and 5)

## 4.1 Formal Language

This section formalizes an object-oriented language with protocol specifications. We briefly introduce expression and class declaration syntax before defining state spaces, access permissions, and permission-based specifications. Finally, we discuss handling of inheritance and enforcement of behavioral subtyping.

### 4.1.1 Syntax

Figure 4.1 shows the syntax of a simple class-based object-oriented language. The language is inspired by Featherweight Java (FJ, Igarashi et al., 1999); we will extend it to include field assignments and typestate protocols in the following subsections. We identify classes ($C$), methods ($m$), and fields ($f$) with their names. As usual, $x$ ranges over variables including the distinguished variable *this* for the receiver object. We use an overbar notation to abbreviate a list of elements. For example, $\overline{x : T}$ stands for $x_1{:}T_1, \ \ldots, \ x_n{:}T_n$. Types ($T$) in our system include Booleans (`bool`) and classes.

Programs are defined with a list of class declarations and a main expression. A class declaration $CL$ gives the class a unique name $C$ and defines its fields, methods, typestates, and state invariants. A constructor is implicitly defined with the class's own and inherited fields. Fields ($F$) are declared with their name and type. Each field is mapped into a part of the state space $n$ that can depend on the field content (details in Section 4.2.2). A method ($M$) declares its result type, formal parameters, specification, and a body expression. State refinements $R$ will be explained in the next section; method specifications $MS$ and state invariants $N$ are deferred to Section 4.1.4.

We syntactically distinguish pure terms $t$ and possibly effectful expressions $e$. Arguments to method calls and object construction are restricted to terms. This simplifies reasoning about effects (Mandelbaum et al., 2003; Chin et al., 2005b) by making execution order explicit.

$$
\begin{array}{rrcl}
\textit{programs} & PR & ::= & \langle \overline{CL}, e \rangle \\
\textit{class decl.} & CL & ::= & \texttt{class C extends C}' \ \{\ \overline{\texttt{F}}\ \overline{\texttt{R}}\ \texttt{I}\ \overline{\texttt{N}}\ \overline{\texttt{M}}\ \} \\
\textit{field decl.} & F & ::= & f : T\ \texttt{in}\ n \\
\textit{meth. decl.} & M & ::= & T\ m(\overline{T\ x}) : MS = e \\
\textit{state decl.} & R & ::= & d = \overline{s}\ \texttt{refines}\ s_0 \\
\textit{terms} & t & ::= & x \mid o \mid \texttt{true} \mid \texttt{false} \\
 & & \mid & t_1\ \texttt{and}\ t_2 \mid t_1\ \texttt{or}\ t_2 \mid \texttt{not}\ t \\
\textit{expressions} & e & ::= & t \mid f \mid \texttt{assign}\ f := t \\
 & & \mid & \texttt{new}\ C(\bar{t}) \mid t_0.m(\bar{t}) \mid \texttt{super}.m(\bar{t}) \\
 & & \mid & \texttt{if}(t, e_1, e_2) \mid \texttt{let}\ x = e_1\ \texttt{in}\ e_2 \\
\textit{values} & v & ::= & o \mid \texttt{true} \mid \texttt{false} \\
\textit{references} & r & ::= & x \mid f \mid o \\
\textit{types} & T & ::= & C \mid \texttt{bool} \\
\textit{nodes} & n & ::= & s \mid d \\
\textit{assumptions} & A & ::= & n \mid A_1 \otimes A_2 \mid A_1 \oplus A_2
\end{array}
$$

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| *classes* | $C$ | *fields* | $f$ | *variables* | $x$ | *objects* | $o$ |
| *methods* | $m$ | *states* | $s$ | *dimensions* | $d$ | | |

Figure 4.1: Core language syntax. Specifications (*I, N, MS*) are in Figure 4.3.

Notice that we syntactically restrict field accesses to fields of the receiver class. Explicit "getter" and "setter" methods can be defined to give other objects access to fields. Assignments replace the previous value of a field with the given one and evaluate to the *previous* field value.[1]

### 4.1.2 State Spaces

State spaces are formally defined as a list of state refinements (see Figure 4.1). A state refinement ($R$) refines an existing state in a new dimension with a set of mutually exclusive sub-states (these concepts are informally introduced in Chapter 3). We use $s$ and $d$ to range over state and dimension names, respectively. A *node* $n$ in a state space can be a state or dimension. State refinements are inherited by subclasses. We assume a root state alive that is defined in the root class `Object`.

We define a variety of helper judgments for state spaces in Figure 4.2. $\mathsf{refinements}(C)$ determines the list of state refinements available in class $C$. $C \vdash A\ \mathsf{wf}$ defines well–formed state assumptions. Assumptions $A$ combine states and are defined in Figure 4.3. Conjunctive assumptions have to cover orthogonal parts of the state space. $C \vdash n \leq n'$ defines the substate relation for a class. $C \vdash A \# A'$ defines orthogonality of state assumptions. $A$ and $A'$ are orthogonal if they refer to different (orthogonal) state dimensions. $C \vdash A \prec n$ defines that a state assumption $A$ only refers to states underneath a root node $n$. $C \vdash A \ll n$ finds the tightest such $n$.

[1] This semantics is appealing in the context of linear type theory, but our tool uses the conventional Java semantics for assignments (cf. Chapter 6).

$$\frac{}{\mathsf{refinements}(\texttt{Object}) = \cdot} \qquad \frac{\texttt{class}\ C\ \texttt{extends}\ C'\ \{\ \overline{F}\ \overline{R}\ \ldots\ \} \qquad \mathsf{refinements}(C') = \overline{R'}}{\mathsf{refinements}(C) = \overline{R'}, \overline{R}}$$

$$\frac{n\ \textit{in}\ \mathsf{refinements}(C)}{C \vdash n\ \mathsf{wf}} \qquad \frac{C \vdash A_1\ \mathsf{wf} \qquad C \vdash A_2\ \mathsf{wf}}{C \vdash A_1 \oplus A_2\ \mathsf{wf}} \qquad \frac{C \vdash A_1\ \mathsf{wf} \qquad A_1\ \#\ A_2 \qquad C \vdash A_2\ \mathsf{wf}}{C \vdash A_1 \otimes A_2\ \mathsf{wf}}$$

$$\frac{d = \overline{s}\ \texttt{refines}\ s \in \mathsf{refinements}(C)}{C \vdash s_i \leq d \qquad C \vdash d \leq s} \qquad \frac{C \vdash n\ \mathsf{wf}}{C \vdash n \leq n} \qquad \frac{C \vdash n \leq n'' \qquad C \vdash n'' \leq n'}{C \vdash n \leq n'}$$

$$\frac{d = \overline{s}\ \texttt{refines}\ s^* \in \mathsf{refinements}(C) \qquad d' = \overline{s'}\ \texttt{refines}\ s^* \in \mathsf{refinements}(C) \qquad d \neq d'}{C \vdash d\ \#\ d'}$$

$$\frac{C \vdash n_1 \leq n_1' \qquad C \vdash n_1'\ \#\ n_2' \qquad C \vdash n_2 \leq n_2'}{C \vdash n_1\ \#\ n_2} \qquad \frac{C \vdash A'\ \#\ A}{C \vdash A\ \#\ A'} \qquad \frac{C \vdash A_1\ \#\ A \qquad C \vdash A_2\ \#\ A}{C \vdash A_1 \otimes A_2\ \#\ A}$$

$$\frac{C \vdash A_1\ \#\ A \qquad C \vdash A_2\ \#\ A}{C \vdash A_1 \oplus A_2\ \#\ A} \qquad \frac{C \vdash n' \leq n}{C \vdash n' \prec n} \qquad \frac{C \vdash A_{1,2} \prec n \qquad C \vdash A_1 \otimes A_2\ \mathsf{wf}}{C \vdash A_1 \otimes A_2 \prec n}$$

$$\frac{C \vdash A_{1,2} \prec n \qquad C \vdash A_1 \oplus A_2\ \mathsf{wf}}{C \vdash A_1 \oplus A_2 \prec n} \qquad \frac{C \vdash A \prec n \qquad \forall n' : C \vdash A \prec n'\ \textit{implies}\ n \leq n'}{C \vdash A \ll n}$$

Figure 4.2: State space judgments (assumptions $A$ defined in Figure 4.1)

### 4.1.3 Access Permissions

Access permissions $p$ give references permission to access an object. Permissions to objects are written $\mathsf{access}(r, n, g, k, A)$ (Figure 4.3). We wrote *perm*$(r, n, g)$ **in** $A$, where *perm* $\in \{\mathsf{unique}, \mathsf{full}, \mathsf{share}, \mathsf{immutable}, \mathsf{pure}\}$, in the previous chapter. The additional parameter $k$ allows us to uniformly represent all permissions as explained below.

- Permissions are granted to references $r$. References can be variables, run-time locations, and fields.
- Permissions apply to a particular *subtree* in the space space of $r$ that is identified by its root node $n$. It represents a *state guarantee* (Section 3.2). Other parts of the state space are unaffected by the permission.
- The *fraction function* $g$ tracks for each node on the path from $n$ to $\mathsf{alive}$ a symbolic fraction (Boyland, 2003). The fraction function keeps track of how often permissions were split at different nodes in the state space so they can be coalesced later (see Section 4.2.5).
- The *below fraction* $k$ encodes the level of access granted by the permission. $k > 0$ grants modifying access. $k < 1$ implies that other potentially modifying permissions exist. Fraction variables $z$ are

conservatively treated as a value between 0 and 1, i.e., $0 < z < 1$.

- An *state assumption* $A$ expresses state knowledge within the permission's subtree. Only full permissions can permanently make state assumptions until they modify the object's state themselves. For weak permissions, the state assumption is *temporary*, i.e., lost after any effectful expression (because the object's state may change without the knowledge of $r$).

We can encode unique, full, share, and pure permissions as follows.

$$\begin{aligned}
\mathsf{unique}(r,n,g)\ \mathbf{in}\ A &\equiv \mathsf{access}(r,n,\{g,n\mapsto 1\},1,A)\\
\mathsf{full}(r,n,g)\ \mathbf{in}\ A &\equiv \mathsf{access}(r,n,g,1,A)\\
\mathsf{share}(r,n,g,k)\ \mathbf{in}\ A &\equiv \mathsf{access}(r,n,g,k,A) \qquad (0<k<1)\\
\mathsf{pure}(n,n,g)\ \mathbf{in}\ A &\equiv \mathsf{access}(r,n,g,0,A)
\end{aligned}$$

One way of thinking about this encoding is that we have two kinds of fractions in a permission: (a) fractions that keeps track of *all* splits and must be strictly positive, and (b) one fraction that keeps track of all splits except when pure permissions are split off. Kind (b) of fractions is set to 0 in pure permissions. This is needed to distinguish unique and full: if kind (a) is 1, we have a unique; if (b) is 1 we have a full. Kind (a) of fractions is kept separately for different nodes in the state hierarchy in the "fraction function"; (b) is the "below fraction". Thus, the fraction function prevents against "loosing track" of pure permissions.

In our formal treatment we omit immutable permissions, but it is straightforward to encode them with an additional "bit" that distinguishes immutable and share permissions. The encoding for unique permits separate unique permissions for orthogonal state dimensions, which is more fine-grained "unique" access than a conventional linear reference to an entire object (Wadler, 1990).

### 4.1.4 Permission-Based Specifications

We combine atomic permissions ($p$) and facts about Boolean values ($q$) using linear logic connectives (Figure 4.3). We also include existential ($\exists z : H.P$) and universal quantification of fractions ($\forall z : H.P$) to alleviate programmers from writing concrete fraction functions in most cases. We type all expressions as an existential type ($E$).

**Method specifications.** Methods are specified with a linear implication ($\multimap$) of predicates ($MS$). The left-hand side of the implication (method pre-condition) may refer to method receiver and formal parameters. The right-hand side (post-condition) existentially quantifies the method result (a similar technique is used in Vault (DeLine and Fähndrich, 2001)). We refer to the receiver with *this* and call the return value *result*.

**State invariants.** We decided to use linear logic predicates for state invariants as well ($N$). In general, several of the defined state invariants will have to be satisfied at the same time. This is due to our hierarchical state spaces. Each class declares an initialization predicate together with a start state ($I$) that are used for object construction (instead of an explicit constructor). The initialization predicate must satisfy the start state's invariant.

$$
\begin{array}{rrcl}
\textit{permissions} & p & ::= & \mathtt{access}(r, n, g, k, A) \\
\textit{facts} & q & ::= & t = \mathtt{true} \mid t = \mathtt{false} \\
\textit{fraction fct.} & g & ::= & z \mid \overline{n \mapsto k} \\
 & & \mid & g/2 \mid g_1, g_2 \\
\textit{fractions} & k & ::= & 1 \mid 0 \mid z \mid k/2 \\
\textit{predicates} & P & ::= & p \mid q \\
 & & \mid & P_1 \otimes P_2 \mid \mathbf{1} \\
 & & \mid & P_1 \,\&\, P_2 \mid \top \\
 & & \mid & P_1 \oplus P_2 \mid \mathbf{0} \\
 & & \mid & \exists z : H.P \mid \forall z : H.P \\
\textit{method specs} & MS & ::= & P \multimap E \\
\textit{expr. types} & E & ::= & \exists x : T.P \\
\textit{state inv.} & N & ::= & n = P \\
\textit{initial state} & I & ::= & \mathtt{initially}\ \langle P, s_1 \otimes \ldots \otimes s_n \rangle \\
\textit{fract. terms} & h & ::= & g \mid k \\
\textit{fract. types} & H & ::= & \mathsf{Fract} \mid \overline{n} \to \mathsf{Fract} \\
\textit{fract. vars.} & z & &
\end{array}
$$

Figure 4.3: Permission-based specifications (using syntactic forms from Figure 4.1)

### 4.1.5 Handling Inheritance

Recall from Section 3.2.2 that permissions give access to a particular frame, usually the virtual frame of an object. Because of subtyping, the precise frame referenced by an object permission is statically unknown. We will call permissions to the virtual frame *object permissions*. Because it is unknown what frame an object permission gives access to, and different frames can be in different states (see Section 3.2.2), we cannot use object permissions to access fields.

$$\textit{references} \quad r \quad ::= \quad \ldots \mid \mathtt{super} \mid \mathtt{this}_{\mathsf{fr}}$$

In order to handle inheritance, we distinguish references to the receiver's "current" frame ($\mathit{this}_{\mathsf{fr}}$) and its super-frame (*super*). Permissions for these "special" references are called *frame permissions*.

A $\mathit{this}_{\mathsf{fr}}$ permission grants access to the fields in the current frame and can be used in method specifications. $\mathit{this}_{\mathsf{fr}}$ permissions cannot be used to call methods. On the other hand, object permissions can be used for calling methods specified with $\mathit{this}_{\mathsf{fr}}$ permissions. In order for this coercion to work, all methods requiring a $\mathit{this}_{\mathsf{fr}}$ permission must be overridden. This guarantees that object permissions used in calling such methods are used by the method to access fields in the virtual frame, which the object permission gives access to. Permissions for `super` are needed for super-calls and are only available in state invariants. Note that for simplicity, all fields are treated as "private": they can only be accessed from inside the class that declares them.

### 4.1.6 Behavioral Subtyping

Subclasses should be allowed to define their own specifications, e.g. to add precision or support additional behavior (Bierhoff and Aldrich, 2005). However, subclasses need to be *behavioral subtypes* (Liskov and

$$\frac{(z : H) \in \Gamma}{\Gamma \vdash z : H} \qquad \frac{}{\Gamma \vdash 1 : \mathsf{Fract}} \qquad \frac{}{\Gamma \vdash 0 : \mathsf{Fract}} \qquad \frac{\Gamma \vdash k : \mathsf{Fract}}{\Gamma \vdash k/2 : \mathsf{Fract}}$$

$$\frac{\Gamma \vdash \overline{k} : \mathsf{Fract} \qquad (k \neq 0)}{\Gamma \vdash \overline{n \mapsto k} : \overline{n} \to \mathsf{Fract}} \qquad \frac{\Gamma \vdash g : \overline{n} \to \mathsf{Fract}}{\Gamma \vdash g/2 : \overline{n} \to \mathsf{Fract}} \qquad \frac{\Gamma \vdash g : \overline{n} \to \mathsf{Fract} \qquad \Gamma \vdash g' : \overline{n'} \to \mathsf{Fract}}{\Gamma \vdash g, g' : \overline{n}, \overline{n'} \to \mathsf{Fract}}$$

$$\frac{\begin{array}{c}\Gamma \vdash r : C \\ C \vdash A \prec n \qquad \Gamma \vdash g : \overline{n} \to \mathsf{Fract} \qquad \overline{n} = \{\textit{all nodes between alive and } n \textit{ inclusive}\} \qquad \Gamma \vdash k : \mathsf{Fract}\end{array}}{\Gamma \vdash \mathsf{access}(r, n, g, k, A)\ \ \mathsf{wf}}$$

Figure 4.4: Fraction typing and well–formed permissions

$$\frac{(x : T) \in \Gamma}{\Gamma \vdash x : T}\ \text{T-VAR} \qquad \frac{(o : C) \in \Gamma}{\Gamma \vdash o : C}\ \text{T-LOC} \qquad \frac{}{\Gamma \vdash \texttt{true} : \texttt{bool}}\ \text{T-TRUE} \qquad \frac{}{\Gamma \vdash \texttt{false} : \texttt{bool}}\ \text{T-FALSE}$$

$$\frac{\Gamma \vdash t_1 : \texttt{bool} \qquad \Gamma \vdash t_2 : \texttt{bool}}{\Gamma \vdash t_1\ \texttt{and}\ t_2 : \texttt{bool}}\ \text{T-AND} \qquad \frac{\Gamma \vdash t_1 : \texttt{bool} \qquad \Gamma \vdash t_2 : \texttt{bool}}{\Gamma \vdash t_1\ \texttt{or}\ t_2 : \texttt{bool}}\ \text{T-OR} \qquad \frac{\Gamma \vdash t : \texttt{bool}}{\Gamma \vdash \texttt{not}\ t : \texttt{bool}}\ \text{T-NOT}$$

$$\frac{\Gamma \vdash t : C' \qquad C' \ \mathsf{extends}\ C}{\Gamma \vdash t : C}\ \text{T-SUB}$$

Figure 4.5: Term typechecking

Wing, 1994) of the extended class. Our system enforces behavioral subtyping in two steps. Firstly, state space inheritance conveniently guarantees that states of subclasses *always* correspond to states defined in superclasses (Bierhoff and Aldrich, 2005). Secondly, we make sure that every overriding method's specification implies the overridden method's specification (Bierhoff and Aldrich, 2005) using the override judgment (Figure 4.7) that is used in checking method declarations. This check leads to method specifications that are contra-variant in the domain and co-variant in the range as required by behavioral subtyping.

## 4.2 Modular Typestate Verification

This section describes a static modular typestate checking technique for access permissions similar to conventional typechecking. It guarantees at compile-time that protocol specifications will never be violated at run-time. We emphasize that our approach does not require tracking typestates at run-time.

$$\textsc{P-Term}\ \frac{\Gamma \vdash t : T \qquad \Gamma; \Delta \vdash [t/x]P}{\Gamma; \Delta \vdash^0 t : \exists x : T.P} \qquad \textsc{P-Field}\ \frac{\mathsf{localFields}(C) = \overline{f : T} \qquad \Gamma; \Delta \vdash [f_i/x]P}{\Gamma; \Delta \vdash^0_C f_i : \exists x : T_i.P}$$

$$\textsc{P-New}\ \frac{\Gamma \vdash \overline{t : T} \qquad \mathsf{init}(C) = \langle \forall \overline{f : T}.P, A \rangle \qquad \Gamma; \Delta \vdash [\overline{t/f}]P}{\Gamma; \Delta \vdash^0 \mathtt{new}\ C(\overline{t}) : \exists x : C.\mathsf{access}(x, \mathsf{alive}, \{\mathsf{alive} \mapsto 1\}, 1, A)}$$

$$\textsc{P-If}\ \frac{(\Gamma, t = \mathtt{true}); \Delta \vdash^i e_1 : \exists x : T.P_1 \setminus \mathcal{E}_1 \qquad \Gamma \vdash t : \mathtt{bool} \qquad (\Gamma, t = \mathtt{false}); \Delta \vdash^j e_2 : \exists x : T.P_2 \setminus \mathcal{E}_2}{\Gamma; \Delta \vdash^{i \vee j} \mathtt{if}(t, e_1, e_2) : \exists x : T.P_1 \oplus P_2 \setminus \mathcal{E}_1 \cup \mathcal{E}_2}$$

$$\textsc{P-Let}\ \frac{\begin{array}{c}\Gamma; \Delta \vdash^i e_1 : \exists x : T.P \setminus \mathcal{E}_1 \qquad (\Gamma, x : T); (\Delta', P) \vdash^j e_2 : E_2 \setminus \mathcal{E}_2 \\ i = 1 \textit{ implies no temporary assumptions in } \Delta' \qquad \textit{Fields in } \mathcal{E}_1 \textit{ do not occur in } \Delta'\end{array}}{\Gamma; (\Delta, \Delta') \vdash^{i \vee j} \mathtt{let}\ x = e_1\ \mathtt{in}\ e_2 : E_2 \setminus \mathcal{E}_1 \cup \mathcal{E}_2}$$

$$\textsc{P-Meth}\ \frac{\begin{array}{c}(\overline{x : T}, \mathtt{this} : C); P \vdash^i_C e : \exists \textit{result} : T_r.P_r \otimes \top \setminus \mathcal{E} \\ E = \exists \textit{result} : T_r.P_r \qquad \mathsf{override}(m, C, \forall \overline{x : T}.P \multimap E)\end{array}}{T_r\ m(\overline{T\ x}) : P \multimap E = e\ \mathsf{ok\ in}\ C}$$

$$\textsc{P-Class}\ \frac{\ldots \qquad \overline{M}\ \mathsf{ok\ in}\ C \qquad \overline{M} \textit{ overrides all methods with this}_{\mathit{fr}} \textit{ permissions in } C'}{\mathtt{class\ C\ extends\ C'}\ \{\ \overline{\mathtt{F}}\ \overline{\mathtt{R}}\ \mathtt{I}\ \overline{\mathtt{N}}\ \overline{\mathtt{M}}\ \}\ \mathsf{ok}}$$

$$\textsc{P-Prog}\ \frac{\overline{CL}\ \mathsf{ok} \qquad \cdot; \cdot \vdash^i_{\_} e : E \setminus \mathcal{E}}{\langle \overline{CL}, e \rangle : E}$$

Figure 4.6: Permission checking for expressions (part 1) and declarations (helper judgments in Figure 4.7)

### 4.2.1 Permission Tracking

We permission-check an expression $e$ with the judgment $\Gamma; \Delta \vdash^{i}_{C} e : \exists x : T.P \setminus \mathcal{E}$. This is read as, "in valid context $\Gamma$ and linear context $\Delta$, an expression $e$ executed within receiver class $C$ has type T, yields permissions $P$, and affects fields $\mathcal{E}$". Permissions $\Delta$ are consumed in the process. We omit the receiver $C$ where it is not required for checking a particular syntactic form. The set $\mathcal{E}$ keeps track of fields that were assigned to, which is important for the correct handling of permissions to fields. It is omitted when empty. The marker $i$ in the judgment can be 0 or 1 where $i = 1$ indicates that states of objects in the context could change during evaluation of the expression. This will help us reason about temporary state assumptions. A combination of markers with $i \vee j$ is 1 if at least one of the markers is 1.

$$\begin{array}{rrcl} \textit{valid contexts} & \Gamma & ::= & \cdot \mid \Gamma, x : T \mid \Gamma, o : C \mid \Gamma, z : H \mid \Gamma, q \\ \textit{linear contexts} & \Delta & ::= & \cdot \mid \Delta, P \\ \textit{effects} & \mathcal{E} & ::= & \cdot \mid \mathcal{E}, f \end{array}$$

Valid and linear contexts distinguish valid (so-called persistent) facts ($\Gamma$) from resources ($\Delta$, also called ephemeral facts). Resources are tracked linearly, forbidding their duplication, while valid facts can be used arbitrarily often. (In logical terms, contraction is defined for valid facts only). The valid context types object variables, fraction variables, and location types and keeps track of terms $q$ known to be true or false. Fraction variables are tracked in order to handle fraction quantification correctly. The linear context holds currently available resource predicates.

Fractions and fraction functions are formally typed in Figure 4.4. Note that fraction function types keep track of exactly which nodes are mapped. Permission validity requires that the fraction function of a permission covers exactly the nodes between (and including) the permission's root node and the state space root alive (Figure 4.4).

The judgment $\Gamma \vdash t : T$ types terms (Figure 4.5) and is completely standard. It includes the usual rule for subsumption using nominal subtyping induced by the extends relation. Term typing is used in expression checking.

Our expression checking rules are syntax-directed up to reasoning about permissions. Permission reasoning is deferred to a separate judgment $\Gamma; \Delta \vdash P$ that uses the rules of linear logic to prove the availability of permissions $P$ in a given context. This judgment will be discussed in Section 4.2.5. Permission checking rules for most expressions appear in Figure 4.6 and are described in turn. Packing, method calls, and field assignment are discussed in following subsections. Helper judgments are summarized in Figure 4.7. The notation $[e'/x]e$ substitutes $e'$ for occurrences of $x$ in $e$.

- P-TERM embeds terms. It formalizes the standard logical judgment for existential introduction and has no effect on existing objects.
- P-FIELD checks field accesses analogously.
- P-NEW checks object construction. The parameters passed to the constructor have to satisfy initialization predicate $P$ and become the object's initial field values. The new existentially quantified object is associated with a unique permission to the root state that makes state assumptions according to the declared start state $A$. Object construction has no effect on existing objects.

  The judgment init (Figure 4.7) looks up initialization predicate and start state for a class. The start state is a conjunction of states (Figure 4.3). The initialization predicate must be strong enough to

$$\frac{\texttt{class}\ C\ \texttt{extends}\ C'\ \{\ldots\} \in \overline{CL}}{C\ \mathsf{extends}\ C'}$$

$$\frac{\texttt{class}\ C\ \{\ldots\ \overline{M}\ \ldots\} \in \overline{CL} \qquad T_r\ m(\overline{T\ x}) : P \multimap \exists \mathit{result} : T_r.P' = e \in \overline{M}}{\mathsf{mtype}(m, C) = \forall \overline{x : T}.P \multimap \exists \mathit{result} : T_r.P'}$$

$$\frac{C\ \mathsf{extends}\ C' \qquad \mathsf{mtype}(m, C') = \forall \overline{x : T}.MS'\ \mathit{implies}\ (\overline{x : T}, \texttt{this} : C); \cdot \vdash MS \multimap MS'}{\mathsf{override}(m, C, \forall \overline{x : T}.MS)}$$

$$\frac{\texttt{class}\ C \ldots\ \{\overline{F}\ \ldots\} \in \overline{CL}}{\mathsf{localFields}(C) = \overline{F}} \qquad \frac{}{\mathsf{init}(\texttt{Object}) = (\mathbf{1}, \mathsf{alive})}$$

$$\frac{\begin{array}{c}\texttt{class}\ C\ \texttt{extends}\ C'\ \{\overline{f : T\ \texttt{in}\ n}\ \overline{S}\ \texttt{initially}\ \langle P' \otimes P, A\rangle\ \ldots\} \\ \mathsf{init}(C') = (\forall \overline{f' : T'}.P', A') \qquad \cdot; (P, \mathsf{access}(\mathit{super}, \mathsf{alive}, \{\mathsf{alive} \mapsto 1\}, A')) \vdash \mathsf{inv}_C(\mathsf{alive}, A) \otimes \top\end{array}}{\mathsf{init}(C) = \langle \forall \overline{f' : T'}, \overline{f : T}.P' \otimes P, A\rangle}$$

$$\frac{\texttt{class}\ C\ \{\ldots\ n = P \ldots\} \in \overline{CL}}{\mathsf{pred}_C(n) = P} \qquad \frac{P = \bigotimes_{n' \le n'' < n} \mathsf{pred}_C(n'')}{\mathsf{pred}_C(n', n) = P}$$

$$\frac{\mathsf{inv}_C(A) = P \Rightarrow n'}{\mathsf{inv}_C(n, A) = P \otimes \mathsf{pred}_C(n', n) \otimes \mathsf{pred}_C(n)} \qquad \frac{}{\mathsf{inv}_C(n) = \mathbf{1} \Rightarrow n}$$

$$\frac{\mathsf{inv}_C(A_i) = P_i \Rightarrow n_i \qquad \mathsf{pred}_C(n_i, n) = P_i' \qquad n_1 \otimes n_2 \ll n \qquad (i = 1, 2)}{\mathsf{inv}_C(A_1 \otimes A_2) = P_1 \otimes P_1' \otimes P_2 \otimes P_2' \Rightarrow n}$$

$$\frac{\mathsf{inv}_C(A_i) = P_i \Rightarrow n_i \qquad \mathsf{pred}_C(n_i, n) = P_i' \qquad n_1 \oplus n_2 \ll n \qquad (i \in 1, 2)}{\mathsf{inv}_C(A_1 \oplus A_2) = (P_1 \otimes P_1') \oplus (P_2 \otimes P_2') \Rightarrow n}$$

$$\frac{\textit{only pure permissions in } P}{\mathsf{effectsAllowed}(P) = 0} \qquad \frac{\textit{exists share, full, or unique permission in } P}{\mathsf{effectsAllowed}(P) = 1}$$

Figure 4.7: Protocol verification helper judgments

prove the invariant needed for the start state. $\mathsf{inv}_C(n, A)$ constructs that invariant for root node $n$ (here, alive) and state information $A$.

- P-IF makes sure that the conditional term is of Boolean type and then assumes its truth (falsehood) in checking the *then* (*else*) branch. This approach lets branches make use of the tested condition. The disjunction used for typing *if* expressions reflects that we statically do not know which branch will be taken. Instead, the condition determines this choice at run-time.

- P-LET checks a `let` binding and represents existential elimination, complementing P-TERM. The linear context used in checking the second subexpression must not mention fields affected by the first expression. This makes sure that outdated field permissions do not "survive" assignments or packing. Moreover, temporary state information is dropped if the first subexpression has side effects.

A program consists of a list of classes and a main expression (P-PROG, Figure 4.6). As usual, the class table $\overline{CL}$ is globally available. The main expression is checked with initially empty contexts. The judgment $CL$ ok (P-CLASS) checks a class declaration. It checks fields, states, and invariants for syntactic correctness in the obvious way (omitted in the rule) and verifies consistency between method specifications and implementations using the judgment $M$ ok in $C$. P-METH assumes the specified pre-condition of a method (i.e. the left-hand side of the linear implication) and verifies that the method's body expression produces the declared post-condition (i.e. the right-hand side of the implication). Conjunction with $\top$ drops excess permissions, e.g., to temporary objects. The override judgment concisely enforces behavioral subtyping (see Section 4.1.6). A method itself is not a linear resource since all resources it uses (including the receiver) are passed in upon invocation; method declaration checking therefore corresponds to universal introduction.

### 4.2.2 Packing and Unpacking

We use a refined notion of *unpacking* (DeLine and Fähndrich, 2004b) to gain access to fields: we unpack and pack a specific permission. Unpacking means to replace a permission for the receiver object with permissions for its fields implied by the invariant of the receiver's current state; packing does the opposite. The access to fields we gain from unpacking reflects the permission we unpacked: full and share permissions give modifying access, while a pure permission gives read-only access to underlying fields.

Packing helps dealing with transitions between scopes (methods), where problems can occur when the same fields are accessed from different scopes, in particular in the presence of reentrancy. To avoid inconsistencies, objects are always fully packed when methods are called. To simplify the situation, only one permission can be unpacked at the same time. Intuitively, we "focus" (Fähndrich and DeLine, 2002) on that permission. This lets us unpack share like full permissions, gaining full rather than shared access to underlying fields (if available). The syntax for packing and unpacking is as follows.

$$\begin{array}{rrcll} \textit{expressions} & e & ::= & \ldots & \mid \ \texttt{unpack}(n, k, A)\ \texttt{in}\ e \\ & & & & \mid \ \texttt{pack to}\ A\ \texttt{in}\ e \end{array}$$

Packing and unpacking always affects the receiver of the currently executed method. The `unpack` parameters express the programmer's expectations about the permission being unpacked. For simplicity, an explicit subtree fraction $k$ is part of `unpack` expressions. It could be inferred from a programmer-provided permission kind, e.g. share.

$$\begin{aligned}
\mathsf{inv}_C(n, g, k, A) &= \mathsf{inv}_C(n, A) \otimes \mathsf{purify}(\mathsf{above}_C(n)) \\
\mathsf{inv}_C(n, g, 0, A) &= \mathsf{purify}\,(\mathsf{inv}_C(n, A) \otimes \mathsf{above}_C(n)) \\
\textit{where } \mathsf{above}_C(n) &= \bigotimes_{n':n<n'\leq \mathsf{alive}} \mathsf{pred}_C(n')
\end{aligned}$$

Figure 4.8: Invariant construction (purify in Figure 4.10)

**Typechecking.** In order for `pack` to work properly we have to "remember" the permission we unpacked. Therefore we introduce unpacked as an additional linear predicate.

$$\textit{permissions} \quad p \quad ::= \quad \ldots \mid \mathsf{unpacked}(n, g, k, A)$$

The checking rules for packing and unpacking are given in Figure 4.9. Notice that packing and unpacking always affects permissions to $\texttt{this}_{\mathsf{fr}}$. (We ignore substitution of `this` with an object location at runtime here.)

P-UNPACK first derives the permission to be unpacked. The judgment inv determines a predicate for the receiver's fields based on the permission being unpacked. It is used when checking the body expression. An unpacked predicate is added into the linear context. We can prevent multiple permissions from being unpacked at the same time using a straightforward dataflow analysis (omitted here).

P-PACK does the opposite of P-UNPACK. It derives the predicate necessary for packing the unpacked permission and then assumes that permission in checking the body expression. The new state assumption $A$ can differ from before only if a modifying permission was unpacked. Finally, the rule ensures that permissions to fields do not "survive" packing.

**Invariant transformation.** The judgment $\mathsf{inv}_C(n, g, k, A)$ determines what permissions to fields are implied by a permission $\mathsf{access}(\textit{this}_{\mathsf{fr}}, n, g, k, A)$ for a frame of class $C$. It is defined in Figure 4.8 and uses $\mathsf{inv}_C(n, A)$ (which we saw in Figure 4.7) to look up declared invariants and a purify function (Figure 4.10) to convert arbitrary permissions into pure permissions.

Unpacking a full or shared permission with root node $n$ yields purified permissions for nodes "above" $n$ and includes invariants following from state assumptions as–is. Conversely, unpacking a pure permission yields completely purified permissions.

### 4.2.3 Calling Methods

Checking a method call involves proving that the method's pre-condition is satisfied. The call can then be typed with the method's post-condition. The rules for method calls perform universal elimination, complementing P-METH.

Unfortunately, calling a method can result in reentrant callbacks. In order to ensure that objects are consistent when called we require them to be fully packed before method calls. This reflects that aliased objects always have to be prepared for reentrant callbacks.

This is not a limitation because we can always pack to some intermediate state. Notice that such *intermediate packing* obviates the need for adoption while allowing focus (Fähndrich and DeLine, 2002): the intermediate state represents the situation where an adopted object was taken out of the adopting object.

P-UNPACK

$$\frac{\Gamma;\Delta \vdash_C \mathsf{access}(\mathit{this}_{\mathsf{fr}}, n, g, k, A) \quad \textit{receiver packed} \quad k = 0 \textit{ implies } i = 0 \quad \Gamma;(\Delta', \mathsf{inv}_C(n,g,k,A), \mathsf{unpacked}(n,g,k,A)) \vdash^i_C e : E \setminus \mathcal{E}}{\Gamma;(\Delta,\Delta') \vdash^i_C \mathtt{unpack}(n,k,A)\ \mathtt{in}\ e : E \setminus \mathcal{E}}$$

P-PACK

$$\frac{\Gamma;\Delta \vdash_C \mathsf{inv}_C(n,g,k,A) \otimes \mathsf{unpacked}(n,g,k,A') \quad k = 0 \textit{ implies } A = A' \quad \Gamma;(\Delta', \mathsf{access}(\mathit{this}_{\mathsf{fr}}, n, g, k, A)) \vdash^i_C e : E \setminus \mathcal{E} \quad \mathsf{localFields}(C) = \overline{f : T\ \mathtt{in}\ n} \quad \textit{Fields do not occur in } \Delta'}{\Gamma;(\Delta,\Delta') \vdash^i_C \mathtt{pack}\ n\ \mathtt{to}\ A\ \mathtt{in}\ e : E \setminus \overline{f}}$$

P-CALL

$$\frac{\Gamma \vdash t_0 : C_0 \quad \Gamma \vdash \overline{t : T} \quad \Gamma;\Delta \vdash [t_0/\mathtt{this}][t_0/\mathtt{this}_{\mathsf{fr}}][\overline{t}/\overline{x}]P \quad \mathsf{mtype}(m, C_0) = \forall \overline{x : T}.P \multimap E \quad i = \mathsf{effectsAllowed}(P) \quad \textit{receiver packed}}{\Gamma;\Delta \vdash^i t_0.m(\overline{t}) : [t_0/\mathtt{this}][t_0/\mathtt{this}_{\mathsf{fr}}][\overline{t}/\overline{x}]E}$$

P-SUPER

$$\frac{\Gamma \vdash \overline{t : T} \quad \Gamma;\Delta \vdash [\mathtt{super}/\mathtt{this}_{\mathsf{fr}}][\overline{t}/\overline{x}]P \quad C\ \mathsf{extends}\ C' \quad \mathsf{mtype}(m, C') = \forall \overline{x : T}.P \multimap E \quad i = \mathsf{effectsAllowed}(P) \quad \textit{receiver packed}}{\Gamma;\Delta \vdash^i_C \mathtt{super}.m(\overline{t}) : [\mathtt{super}/\mathtt{this}_{\mathsf{fr}}][\overline{t}/\overline{x}]E}$$

P-ASSIGN

$$\frac{\Gamma;\Delta \vdash t : \exists x : T_i.P \quad \Gamma;\Delta' \vdash_C [f_i/x']P' \otimes p \quad \mathsf{localFields}(C) = \overline{f : T\ \mathtt{in}\ n} \quad n_i \leq n \quad p = \mathsf{unpacked}(n,g,k,A), k \neq 0}{\Gamma;(\Delta,\Delta') \vdash^1_C \mathtt{assign}\ f_i := t : \exists x' : T_i.P' \otimes [f_i/x]P \otimes p \setminus f_i}$$

Figure 4.9: Permission checking for expressions (part 2)

$$\frac{p = \mathsf{access}(r,n,g,k,A)}{\mathsf{purify}(p) = \mathsf{pure}(r,n,g,A)} \qquad \frac{\mathsf{purify}(P_1) = P_1' \quad \mathsf{purify}(P_2) = P_2' \quad \mathtt{op} \in \{\otimes, \&, \oplus\}}{\mathsf{purify}(P_1\ \mathtt{op}\ P_2) = P_1'\ \mathtt{op}\ P_2'}$$

$$\frac{\mathtt{unit} \in \{\mathbf{1}, \top, \mathbf{0}\}}{\mathsf{purify}(\mathtt{unit}) = \mathtt{unit}} \qquad \frac{\mathsf{purify}(P) = P'}{\mathsf{purify}(\exists z : H.P) = \exists z : H.P'} \qquad \frac{\mathsf{purify}(P) = P'}{\mathsf{purify}(\forall z : H.P) = \forall z : H.P'}$$

Figure 4.10: Permission purification

**Virtual calls.** Virtual calls are dynamically dispatched (rule P-CALL). In virtual calls, frame and object permissions are identical because object permissions simply refer to the object's virtual frame. This is achieved by substituting the given receiver for both *this* and $\mathit{this}_{\mathsf{fr}}$.

**Super calls.** Super calls are statically dispatched (rule P-SUPER). Recall that `super` is used to identify permissions to the super-frame. We substitute `super` only for $\mathit{this}_{\mathsf{fr}}$. We omit the identity substitution of *this* for the receiver (*this* again) for clarity.

### 4.2.4 Field Assignments

Assignments to fields change the state of the receiver's current frame. We point out that assignments to a field do *not* change states of objects referenced by the field. Therefore reasoning about assignments mostly has to be concerned with preserving invariants of the receiver. The unpacked predicates introduced in Section 4.2.2 help us with this task.

Our intuition is that assignment to a field requires unpacking the surrounding object to the point where all states that refer to the assigned field in their invariants are revealed. Notice that the object does not have to be unpacked completely in this scheme. For simplicity, each field is annotated with the subtree that can depend on it (Figure 4.1). Thus we interpret nodes as data groups (Leino, 1998).

The rule P-ASSIGN (Figure 4.9) assigns a given object $t$ to a field $f_i$ and returns the old field value as an existential $x'$. This preserves information about that value. The rule verifies that the new object is of the correct type and that a suitable full or share permission is currently unpacked. By recording an effect on $f_i$ we ensure that information about the old field value cannot "flow around" the assignment (which would be unsound).

### 4.2.5 Permission Reasoning with Splitting and Joining

Our permission checking rules rely on proving a predicate $P$ given the current valid and linear resources, written $\Gamma; \Delta \vdash P$ (Figure 4.11). We use standard rules for the decidable multiplicative-additive fragment of linear logic (MALL) with quantifiers that only range over fractions (Lincoln and Scedrov, 1994). Following Zhao (2007) we introduce a notion of substitution into the logic that allows substituting a set of linear resources with an equivalent one (SUBST, Figure 4.11), similar to a conventional subtyping rule in non-linear logics.

The judgment $P \Rrightarrow P'$ defines legal substitutions. We use substitutions for splitting and merging permissions (Figure 4.12). The symbol $\Lleftarrow\Rrightarrow$ indicates that transformations are allowed in both directions. SYM and ASYM generalize the rules from Section 3.1. Most other rules are used to split permissions for larger subtrees into smaller ones and vice versa. We explain each rule in turn.

SYM symmetrically splits a permission into two equivalent permissions. Notice how fractions are split. ASYM asymmetrically splits a pure permission off a given permission. Here, the subtree fraction $k$ is untouched, reflecting the asymmetric split. Both transformations can be inverted. We require non-contradicting state information when merging permissions.

F-SPLIT-$\otimes$ splits a full permission with a conjunctive state assumption into a conjunction of full permissions. F-MERGE-$\otimes$ inverts F-SPLIT-$\otimes$ but requires the fraction on the new root node to be 1. This guarantees that no additional full or shared permissions exist in the new permission's subtree. F-$\oplus$ splits

$$\textsc{LinHyp}\ \frac{}{\Gamma; P \vdash P} \qquad \textsc{Subst}\ \frac{\Gamma; \Delta \vdash P' \qquad P' \Rrightarrow P}{\Gamma; \Delta \vdash P}$$

$$\otimes\text{ I}\ \frac{\Gamma; \Delta_1 \vdash P_1 \qquad \Gamma; \Delta_2 \vdash P_2}{\Gamma; (\Delta_1, \Delta_2) \vdash P_1 \otimes P_2} \qquad \otimes\text{ E}\ \frac{\Gamma; \Delta \vdash P_1 \otimes P_2 \qquad \Gamma; (\Delta', P_1, P_2) \vdash P}{\Gamma; (\Delta, \Delta') \vdash P}$$

$$\mathbf{1}\text{ I}\ \frac{}{\Gamma; \cdot \vdash \mathbf{1}} \qquad \mathbf{1}\text{ E}\ \frac{\Gamma; \Delta \vdash \mathbf{1} \qquad \Gamma; \Delta' \vdash P}{\Gamma; (\Delta, \Delta') \vdash P}$$

$$\&\text{ I}\ \frac{\Gamma; \Delta \vdash P_1 \qquad \Gamma; \Delta \vdash P_2}{\Gamma; \Delta \vdash P_1 \,\&\, P_2} \qquad \&E_L\ \frac{\Gamma; \Delta \vdash P_1 \,\&\, P_2}{\Gamma; \Delta \vdash P_1} \qquad \&E_R\ \frac{\Gamma; \Delta \vdash P_1 \,\&\, P_2}{\Gamma; \Delta \vdash P_2}$$

$$\top\text{ I}\ \frac{}{\Gamma; \Delta \vdash \top}$$

*no* $\top$ *elimination*

$$\oplus I_L\ \frac{\Gamma; \Delta \vdash P_1}{\Gamma; \Delta \vdash P_1 \oplus P_2} \qquad \oplus I_R\ \frac{\Gamma; \Delta \vdash P_2}{\Gamma; \Delta \vdash P_1 \oplus P_2}$$

$$\oplus\text{ E}\ \frac{\Gamma; \Delta \vdash P_1 \oplus P_2 \qquad \Gamma; (\Delta', P_1) \vdash P \qquad \Gamma; (\Delta', P_2) \vdash P}{\Gamma; (\Delta, \Delta') \vdash P}$$

*no* $\mathbf{0}$ *introduction*

$$\mathbf{0}\text{ E}\ \frac{\Gamma; \Delta \vdash \mathbf{0}}{\Gamma; (\Delta, \Delta') \vdash P}$$

$$\forall\text{ I}\ \frac{(\Gamma, z : H); \Delta \vdash P}{\Gamma; \Delta \vdash \forall z : H.P} \qquad \forall\text{ E}\ \frac{\Gamma \vdash h : H \qquad \Gamma; \Delta \vdash \forall z : H.P}{\Gamma; \Delta \vdash [h/z]P}$$

$$\exists\text{ I}\ \frac{\Gamma \vdash h : H \qquad \Gamma; \Delta \vdash [h/z]P}{\Gamma; \Delta \vdash \exists z : H.P} \qquad \exists\text{ E}\ \frac{\Gamma; \Delta \vdash \exists z : H.P \qquad (\Gamma, z : H), (\Delta', P) \vdash P'}{\Gamma; (\Delta, \Delta') \vdash P'}$$

Figure 4.11: Affine logic for permission reasoning

and conjoins full permissions with a disjunction of state assumptions. Since only one of the two state assumptions can be true at a given time we do not need to split fractions.

F-DOWN limits a full permission to a smaller subtree by moving the root node down in the state space. The fraction function is appended with additional 1 fractions for nodes that are above the moved root. Notice that this operation is only allowed if the state assumptions of the original permission are inside the new root node (in order to ensure well-formedness of the new permission). F-UP does the opposite but like F-MERGE-$\otimes$ it requires the fraction on the new root node to be 1. Similarly, P-UP can be used to weaken a pure permission by moving its root up in the state space. Finally, FORGET allows a permission to "forget" its state assumption. This rule is used to drop temporary state assumptions and can be used to match state assumptions between permissions prior to applying rules such as SYM, ASYM, and F-DOWN.

Our splitting and joining rules maintain a consistent set of permissions for each object so that no permission can ever violate an assumption another permission makes. Fractions of all permissions to an object sum up to (at most) 1 for every node in the object's state space. These rules encompass the needed kinds of permission manipulations that we are aware of:

- Obtaining a needed *kind* of permission, such as a pure from a full, with symmetric and asymmetric splitting and merging of permissions for a given node (SYM, ASYM).
- Splitting and merging permissions along dimensions (F-SPLIT, F-JOIN).
- Obtaining a permission with a needed root, by moving roots up or down (F-DOWN, F-UP, P-UP).

FORGET could be made to allow dropping *some* state assumptions while keeping others, but we omit this generalization for clarity.

### 4.2.6 Soundness

Bierhoff and Aldrich (2007a) present a proof of soundness for a fragment of the system presented in this dissertation. The fragment does not include inheritance and only supports permissions for objects as a whole. State dimensions are omitted and specifications are deterministic. The fragment does include full, share, and pure permissions with explicit fractions and temporary state information. Pre-conditions, post-conditions, and invariants are conjunctions of such permissions.

Soundness is proven with the canonical progress and preservation theorems. Progress means that a well-typed program is always either already a value or can take an evaluation step. Preservation shows that a well-typed program always remains well-typed when it takes an evaluation step. "Well-typed" here means that permissions and typestates correctly approximate runtime behavior. Therefore, these two theorems together intuitively prove that well-typed programs will never violate protocols declared in the program using permissions and typestates.

### 4.2.7 Example

To illustrate how verification proceeds, Figure 4.13 shows the `fill` method from `BufferedInputStream` (Figure 3.12) written in our core language. As can be seen we need an intermediate state reads and a marker field `reading` that indicate an ongoing call to the underlying stream. We also need an additional

SYM
$$\frac{A = A' = A'' \textit{ or } (A = A' \textit{ and } A'' = n) \textit{ or } (A = A'' \textit{ and } A' = n)}{\mathsf{access}(r, n, g, k, A) \Leftrightarrows \mathsf{access}(r, n, g/2, k/2, A') \otimes \mathsf{access}(r, n, g/2, k/2, A'')}$$

ASYM
$$\frac{A = A' = A'' \textit{ or } (A = A' \textit{ and } A'' = n) \textit{ or } (A = A'' \textit{ and } A' = n)}{\mathsf{access}(r, n, g, k, A) \Leftrightarrows \mathsf{access}(r, n, g/2, k, A') \otimes \mathsf{pure}(r, n, g/2, A'')}$$

F-SPLIT-$\otimes$
$$\frac{n_1 \# n_2 \qquad A_1 \prec n_1 \leq n \qquad A_2 \prec n_2 \leq n \qquad p_i = \mathsf{full}(r, n_i, \{g, \mathsf{nodes}(n_i, n) \mapsto 1\}/2, A_i)}{\mathsf{full}(r, n, g, A_1 \otimes A_2) \Rrightarrow p_1 \otimes p_2}$$

F-MERGE-$\otimes$
$$\frac{n_1 \# n_2 \qquad A_1 \prec n_1 \leq n \qquad A_2 \prec n_2 \leq n \qquad p_i = \mathsf{full}(r, n_i, \{g, n \mapsto 1, \mathsf{nodes}(n_i, n) \mapsto 1\}/2, A_i)}{p_1 \otimes p_2 \Rrightarrow \mathsf{full}(r, n, \{g, n \mapsto 1\}, A_1 \otimes A_2)}$$

F-$\oplus$
$$\frac{A_1 \# A_2}{\mathsf{full}(r, n, g, A_1 \oplus A_2) \Leftrightarrows \mathsf{full}(r, n, g, A_1) \oplus \mathsf{full}(r, n, g, A_2)}$$

F-DOWN
$$\frac{A \prec n' \leq n}{\mathsf{full}(r, n, g, A) \Rrightarrow \mathsf{full}(r, n', \{g, \mathsf{nodes}(n', n) \mapsto 1\}, A)}$$

F-UP
$$\frac{A \prec n' \leq n}{\mathsf{full}(r, n', \{g, n \mapsto 1, \mathsf{nodes}(n', n) \mapsto 1\}, A) \Rrightarrow \mathsf{full}(r, n, \{g, n \mapsto 1\}, A)}$$

P-UP
$$\frac{n' \leq n}{\mathsf{pure}(r, n, \{g, \mathsf{nodes}(n', n) \mapsto \overline{k}\}, A) \Rrightarrow \mathsf{pure}(r, n', g, A)}$$

FORGET
$$\frac{}{\mathsf{access}(r, n, g, k, A) \Rrightarrow \mathsf{access}(r, n, g, k, n)}$$

Figure 4.12: Splitting and merging of access permissions

state refinement to specify an internal method replacing the `while` loop in the original implementation. (We assume that $this_{\mathsf{fr}}$ permissions can be used for calls to `private` methods.)

Maybe surprisingly, we have to reassign field values after `super.read()` returns. The reason is that when calling `super` we lose temporary state information for *this*. Assignment re-establishes this information and lets us pack properly before calling `doFill` recursively or terminating in the cases of a full buffer or a depleted underlying stream.

It turns out that these re-assignments are *not* just an inconvenience caused by our method but point to a real problem in the Java standard library implementation. We could implement a malicious underlying stream that calls back into the "surrounding" `BufferedInputStream` object. This call changes a field, which causes the buffer's invariant on *count* to permanently break, *later on* resulting in an undocumented array bounds exception when trying to read beyond the end of the buffer array.

Because `fill` operates on a share permission our verification approach forces taking into account possible field changes through reentrant calls with other share permissions. (This is precisely what our malicious stream does.) We could avoid field re-assignments by having `read` require a full permission, thereby documenting that reentrant (modifying) calls are not permitted for this method.

```
class BufferedInputStream extends FilterInputStream {
states ready, reads refine open; ...
states partial, complete refine filled;

reads := reading; ready := reading = false; ...

private boolean reading; ...

public int read() : ∀k : Fract. ... =
unpack(open, k) in
  let r = reading in if(r == false, ...  fill() ...  )

private bool fill() : ∀k : Fract.
  share(this_fr, open) in depleted ⊕ eof ⊸
    share(this_fr, open) in available ⊕ eof =
unpack(open, k, depleted ⊕ eof) in
  assign count = 0 in assign pos = 0 in
  assign reading = true in
  pack to reads in
    let b = super.read() in
    unpack(open, k, open) in
      let r = reading in assign reading = false in
      assign count = 0 in assign pos = 0 in
      if(r, if(b = -1, pack to eof in false,
              pack to depleted in doFill(b)),
            pack to eof in false)

private bool doFill(int b) : ∀k : Fract.
  share(this_fr, open) in depleted ⊕ partial ⊸
    share(this_fr, open) in partial ⊕ complete =
unpack(open, k, depleted ⊕ partial) in
  let c = count in let buffer = buf in
  assign buffer[c] = b in assign count = c + 1 in
  let l = buffer.length in
  if(c + 1 >= l, pack to complete in true,
    assign reading = true in pack to reads in
      let b = super.read() in unpack(open, k) in
        let r = reading in assign reading = false in
        assign count = c + 1 in assign pos = 0 in
        pack to partial in
          if(r == false || b == -1, true, doFill(b))
```

Figure 4.13: Fragment of `BufferedInputStream` from Figure 3.12 in core language

# Chapter 5

# Polymorphic Permission Inference

Fractional permissions (Boyland, 2003) have recently received much attention for sound static reasoning about programs that rely on aliasing. They have been used for avoiding data races with locks (Terauchi and Aiken, 2008) as well as for verifying properties of multi-threaded (Bornat et al., 2005; Beckman et al., 2008; Leino and Müller, 2009) and single-threaded programs (this dissertation), where fractional permissions are typically embedded into a substructural logic.

The type system for access permissions presented in the previous chapter is highly non-deterministic (as detailed below). Intuitively, challenges in automating the type system arise from permission splitting: one could potentially have to split permissions an arbitrary number of times before a predicate such as a method pre-condition is provable.

This chapter presents a type inference algorithm for proving permission-based assertions in a decidable fragment of linear logic (Girard, 1987; Lincoln and Scedrov, 1994). Unlike previous work, our inference approach supports *polymorphism* over fractions, i.e., universally and existentially quantified fractions.

The permission inference algorithm presented in this chapter extends conventional resource management techniques for linear logic (Cervesato et al., 2000) to additionally infer how fractional permissions should to be split and merged over time. This process requires collecting linear constraints over fraction variables (Section 5.1)—which is technically similar to previous work by Terauchi and Aiken (2008)—and ensuring that these remain satisfiable (Section 5.2).

## 5.1 Inference System

The permission type system presented in the previous section has several sources of non-determinism; in other words, it makes several kinds of "guesses":

- Context splits, such as in the linear logic proof rule for $\otimes$, "guess" how permissions have to be divided in order to satisfy different predicates. Notice that context splits also appear in the typing rules in order to use different resources to prove the branches of a `let` binding.

- Multiple proof rules can apply. For example, the rules for proving a disjunction $P_1 \oplus P_2$ "guess" whether $P_1$ or $P_2$ can be proven.

- Permissions can be split an arbitrary number of times and merged back together. The type system "guesses" exactly how a given permission has to be split up and merged in order to satisfy different predicates. Notice that this additional non-determinism is not an issue in conventional linear logic proof search because resources are indivisible. However, permission splits intuitively happen together with context splits in order to use the split-up permissions for proving different predicates.

This section presents a type inference algorithm for inferring how permissions flow through a program. This algorithm is based on conventional methods for resource management in linear logic (Cervesato et al., 2000) and adds linear constraints, similar to constraint logic programming (Jaffar and Lassez, 1987), in order to infer permission splits and merges. The constraints essentially "delay" decisions about splits and merges to make sure that permissions needed later on in the program can be satisfied.

Our existing typechecking and proof judgments (ignoring typing information) $\Delta \vdash e : \exists x.P$ and $\Delta \vdash P$ prove a predicate $P$ based on given resources $\Delta$. They essentially treat $\Delta$ as an input and produce one output, the predicate $P$ that could be proven.

In order to manage context splits, resource management techniques for linear logic proof search (Cervesato et al., 2000) add an additional output, the "leftover" resources $\Delta'$, to these judgments. The idea of resource management is that we can make context splits for proving two predicates $P_1$ and $P_2$ deterministic by sending all resources down the first branch (that proves $P_1$) and then use the leftover resources to prove the second predicate $P_2$, such as in the following sample rule for proving multiplicative conjunction:

$$\frac{\Delta \vdash P_1 \Rightarrow \Delta' \qquad \Delta' \vdash P_2 \Rightarrow \Delta''}{\Delta \vdash P_1 \otimes P_2 \Rightarrow \Delta''}$$

We extend this idea by tracking constraints $C$ together with the current linear context $\Delta$ that encode what permissions were needed in the different parts of the program. Again, we need input and output constraints so that different parts of the proof can add their constraints to the existing ones. In fact, resources $\Delta$ will always be paired with their constraints $C$, usually written $\Delta \mid C$. We call such a pair an *atomic context*. Constraints and their collection will be discussed in more detail below.

This leaves one final source of non-determinism: situations where multiple proof rules for linear logic are applicable. This is the case for proving an external choice, $P_1 \oplus P_2$ (there are two right-rules for $\oplus$ in Figure 4.11), and for using an internal choice, $P_1 \mathbin{\&} P_2$ (there are two left-rules for $\&$). Typically, backtracking is used for these cases. Backtracking means that we arbitrarily choose one of the applicable rules, and if that does not work, we try the other. But backtracking is awkward for our purposes: we are trying to typecheck an entire program, which involves proving linear logic predicates for every method call and object construction. Whenever such a proof fails we would have to backtrack to the last call or construction site in the program where we made a choice. Doing so as part of a dataflow analysis (cf. Chapter 6), where we are trying to infer loop invariants as well, is decidedly non-trivial because we would have to roll back the entire state of the flow analysis, including previously computed lattice values and nodes that still have to be analyzed. Therefore, instead of backtracking, we will carry all possible choices forward.

The following sections discuss the syntax, typechecking rules, and proof rules of our permission inference system.

| | | | | |
|---|---|---|---|---|
| *Programs* | $\Lambda$ | ::= | $\bullet$ | |
| | | \| | $\Lambda, m(x_1, \ldots, x_n) : P \multimap E = M$ | *method* |
| | | \| | $\Lambda, C(x_1, \ldots, x_n) : \langle P, A \rangle$ | *constructor* |
| *Expressions* | $M$ | ::= | $x_0.m(x_1, \ldots x_n)$ | *call* |
| | | \| | $\texttt{new}\ C(x_1, \ldots, x_n)$ | *construction* |
| | | \| | $\texttt{let}\ x = M_1\ \texttt{in}\ M_2$ | *sequence* |
| *Expression types* | $E$ | ::= | $\exists x.P$ | |
| *Predicates* | $P$ | ::= | $\mathsf{access}(x, n, g)$ **in** $A$ | *atom* |
| | | \| | $P_1 \otimes P_2$ | |
| | | \| | $P_1 \,\&\, P_2$ | |
| | | \| | $P_1 \oplus P_2$ | |
| | | \| | $\forall z.P$ | |
| | | \| | $\exists z.P$ | |
| *Fraction functions* | $g$ | ::= | $\{n_1 \mapsto k_1, \ldots, n_j \mapsto k_j, \mathsf{below} \mapsto k\}$ | |
| *Fractions* | $k$ | ::= | $z$ | *unknown fraction* |
| | | \| | $\mathbf{Z}$ | *fraction variable* |
| | | \| | $1$ | *one* |
| | | \| | $0$ | *zero* |
| *State information* | $A$ | ::= | $n_1, \ldots, n_j$ | |
| *Variables* | $\Gamma$ | ::= | $\bullet$ | *empty* |
| | | \| | $\Gamma, x$ | *variable* |
| *Contexts* | $\Psi$ | ::= | $\Delta \mid C$ | *atom* |
| | | \| | $\Psi_1 \,\&\, \Psi_2$ | *choice* |
| | | \| | $\Psi_1 \oplus \Psi_2$ | *all* |
| *Permission set* | $\Delta$ | ::= | $\bullet$ | *empty* |
| | | \| | $\Delta, P$ | *extension* |
| *Constraints* | $C$ | ::= | $\top$ | *true* |
| | | \| | $\bot$ | *false* |
| | | \| | $F$ | *formula* |
| | | \| | $C_1 \wedge C_2$ | *conjunction* |
| *Formulae* | $F$ | ::= | $k \doteq k_1 + \ldots + k_j$ | |
| | | \| | $k_1 \leq k_2$ | |
| | | \| | $0 < k$ | |
| *Program variables* | $x, y$ | ::= | *this* \| ... | |
| *Fraction variables* | $z$ | | | |
| *Class names* | $C$ | ::= | `Object` \| ... | |
| *Node names* | $n$ | ::= | `alive` \| ... | |
| *Method names* | $m$ | | | |

Figure 5.1: Syntax for permission inference system

### 5.1.1 Syntax

The syntax for the permission inference system is summarized in Figure 5.1. Like the original type system, it is based on Featherweight Java (Igarashi et al., 1999), but for simplicity we are not keeping track of object types. Instead, we are assuming that conventional typechecking has already ensured type safety. We are also ignoring subtyping in this calculus. Programs $\Lambda$ therefore are simply lists of methods (suitably renamed to avoid name clashes) and constructors. Expressions $M$ are standard; however, notice that we introduce a let-binding construct to define intermediate variables and only allow variables as arguments to methods and object constructors. This lets programs written in this language correspond more directly with our original type system.

Expression types existentially bind a variable that represents the value computed by an expression and simply consist of a linear logic predicate $P$. Predicates are completely standard and include existential and universal quantification of fraction variables as in our original type system. Permissions $\mathsf{access}(x, n, g)$ **in** $A$ represent the only kind of atomic predicate. A fraction function $g$ for a permission $\mathsf{access}(x, n_i, g)$ is written $\{n_1 \mapsto k_1, \ldots, n_i \mapsto k_i, \mathsf{below} \mapsto k\}$, where (implicitly) $n_i \leq \ldots \leq n_1 = \mathsf{alive}$. (Notice that the permission's root matches the last node entry in the fraction function.) It maps nodes $n$ to to *node fractions*; notice that for readability we include the *below fraction* in $g$. Fractions will include unknown fractions $z$ as well as fraction variables $\mathbf{Z}$ which are used during proof search to find a suitable instantiation of quantified variables.

Proof contexts $\Psi$ are built up from atomic contexts $\Delta \mid C$ as described in the previous section. Whenever we can make a choice between two rules we introduce a *choice context* $\Psi_1 \;\&\; \Psi_2$ that carries the two possibilities forward (to avoid backtracking). When one of the choices fails to prove a predicate, we will simply drop it, and if no context remains then our proof search fails, meaning there is a protocol violation in the program. It ends up being convenient to also introduce *all contexts* $\Psi_1 \oplus \Psi_2$ to encode that two possibilities have to be taken into account. This is useful for situations where the same resources need to be used in two different subsequent proofs. Linear contexts $\Delta$ are lists of predicates, and constraints $C$ are conjunctions of linear formulae $F$ over fractions. We will encounter three kinds of formulae, namely equating a fraction to a sum of fractions, relating two fractions, and asserting that a fraction is strictly greater than zero. $\top$ will be used for trivially true constraints, and we will use $\bot$ for unsatisfiable constraints.

### 5.1.2 Typechecking Rules

Expressions are typechecked with the following judgment:

$$\Gamma \mid \Psi \vdash M : \exists x.P \Rightarrow \Psi'$$

This judgment can be read as, in context $\Psi$, expression M (with free variables defined in $\Gamma$) will produce a value $x$ with predicate $P$ as well as a new context $\Psi'$ for the remainder of the program. The typechecking rules are shown in Figure 5.2. They rely on helper judgments shown in Figure 5.3. They also use the proof judgment $\Psi \vdash P \Rightarrow \Psi'$ which will be discussed in the following section. The rules for typechecking expressions are in fact straightforward since constraint collection happens in the proof judgment.

- As before, T-CALL looks up the invoked method's declared signature and renames the parameters, including the receiver, to match the arguments provided. It then proves the method pre-condition in the given context, which will produce a new context that likely contains additional constraints in order

$$\text{T-CALL}\quad \frac{\mathsf{mtype}(m) = \forall x_0, x_1, \ldots, x_n.P \multimap E \qquad \Psi \vdash P \Rightarrow \Psi'}{\Gamma \mid \Psi \vdash x_0.m(x_1, \ldots, x_n) : E \Rightarrow \Psi'}$$

$$\text{T-NEW}\quad \frac{\begin{array}{c}\mathsf{init}(C) = \langle \forall x_1, \ldots, x_n.P, A \rangle \\ \Psi \vdash P \Rightarrow \Psi' \qquad E = \exists x.\mathsf{access}(x, \mathsf{alive}, \{\mathsf{alive} \mapsto 1, \mathsf{below} \mapsto 1\})\ \textbf{in}\ A\end{array}}{\Gamma \mid \Psi \vdash \mathtt{new}\, C(x_1, \ldots, x_n) : E \Rightarrow \Psi'}$$

$$\text{T-LET}\quad \frac{\Gamma \mid \Psi \vdash M_1 : \exists x.P \Rightarrow \Psi' \qquad \Gamma, x \mid \Psi', P \vdash M_2 : E \Rightarrow \Psi''}{\Gamma \mid \Psi \vdash \mathtt{let}\ x = M_1\ \mathtt{in}\ M_2 : E \Rightarrow \Psi''}$$

Figure 5.2: Typechecking rules for permission inference

to ensure the pre-condition's satisfiability. The new context is used as the output context of the method invocation expression, while the method post-condition types the expression.

- Similarly, T-NEW creates a new object of class $C$ by looking up its initialization predicate and proving it with the given parameters. The expression is typed with the new object's unique permission and returns the context resulting from proving the initialization predicate.

- T-LET types let-bindings by first collecting constraints from the first branch and then checking the second branch using the outputs from the first branch.

All typing judgments must be well-formed in order to avoid free variables in permission predicates.

$$\frac{\Gamma \vdash \Psi \qquad \Gamma \vdash M \qquad \Gamma \vdash E \qquad \Gamma \vdash \Psi'}{\Gamma \mid \Psi \vdash M : E \Rightarrow \Psi'}$$

This rule stipulates that the outputs $E$ and $\Psi'$ must only contain free variables defined in $\Gamma$. In particular, this means that the variable defined in a let-binding cannot occur free in the outputs generated when typechecking the binding expression.

We omit typing rules for field accesses and the related packing and unpacking operations since they are orthogonal to the problem of permission inference. Section 6.3.7 will discuss how these can be handled in a tool.

The constraints collected by the typechecking and proof rules are similar in nature to previous work on inferring fractions to guarantee data race freedom (Terauchi and Aiken, 2008; Terauchi, 2008), although our rules are complicated by fraction quantification, the treatment of linear logic, and the presence of fraction functions (instead of an individual fraction per object). Furthermore, we treat control flow differently, as discussed in Chapter 6. Handling universally quantified fractions seems to improve modularity, since each caller of a method can use fractions of their choosing to instantiate universal quantifiers.

$$\frac{m(x_1,\ldots,x_n) : P \multimap E = M \in \Lambda}{\mathsf{mtype}(m) = \forall \mathit{this}, x_1,\ldots,x_n.P \multimap E} \qquad \frac{C(x_1,\ldots,x_n) : \langle P, A\rangle \in \Lambda}{\mathsf{init}(C) = \langle \forall x_1,\ldots,x_n.P, A\rangle}$$

Figure 5.3: Helper judgments for permission inference

**Helper Judgments.** Helper judgments (Figure 5.3) are needed to look up method signatures and object initialization predicates and are defined analogously to the type system.

### 5.1.3 Proof Rules

The judgment for proving a predicate $P$ from an atomic context is the following:

$$\Psi \vdash P \Rightarrow \Psi'$$

These will be presented in two steps, for deriving constraints for an individual permission and for combining them when proving linear logic formulae, given an atomic context $\Delta \mid C$. Additionally we will have context rules that break down contexts until proof rules can be invoked on atomic contexts $\Delta \mid C$.

#### Atom Rules

Figure 5.4 shows the complete set of rules for proving a single permission in a given context. In particular, these rules handle permissions with root nodes coming from a hierarchical state space. We do not model state dimensions here, but it is possible to extend ATOM-MOVE-DOWN and ATOM-MOVE-UP to multiple dimensions, and this is in fact what our tool does (cf. Chapter 6).

There are three rules for splitting permissions and only one for merging permissions. The basic idea behind these rules is to delay root moves until they are needed for splitting off a given permission, while permissions for the same root are merged eagerly. A key invariant for these rules to work is that permissions for the same object but different *related* roots $n_1$ and $n_2$, where $n_1 \leq n_2$, can never occur in this system.[1]

As mentioned in Section 5.1.1, constraint formulae have three forms (Figure 5.1): they equate a fraction sum with another fraction, define a fraction to be strictly greater than 0, or relate two fractions. The first kind of formula is used to relate corresponding fractions from different fraction functions. The other two are used to define well-formedness of fraction functions as explained below.

In order for a fraction function $\{n_1 \mapsto k_1, \ldots, n_i \mapsto k_i, \mathsf{below} \mapsto k\}$ to be well-formed, its fractions have to be monotonically increasing from the fraction $k_1$ for the state space root $n_1 = \mathsf{alive}$ to $k_i$, the fraction for the permission's root node $n_i$. We achieve this with constraints $k_1 \leq \ldots \leq k_i$. (Notice that while $n_1 \leq n_2$ compares nodes in the state space, $k_1 \leq k_2$ compares fractions.) This means that nodes "higher up" in the hierarchy are mapped to a smaller fraction than nodes that refine them. This is because of root moves: fractions for "higher up" nodes, in particular the root node, may be affected by permission

[1] They would have to cover orthogonal dimensions, and our implementation keeps those separate until they have to be merged into a common root node, which is achieved by generalizing ATOM-MOVE-UP. ATOM-MOVE-DOWN also has to be generalized to possibly place multiple leftover permissions into the output context, one for each dimension materialized by the root move.

ATOM-ROOT-MATCH

$$\frac{\begin{array}{c} A \leq A' \qquad g = \{n_1 \mapsto k_1, \ldots, n_j \mapsto k_j, \mathsf{below} \mapsto k\} \\ g' = \{n_1 \mapsto k'_1, \ldots, n_j \mapsto k'_j, \mathsf{below} \mapsto k'\} \qquad g'' = \{n_1 \mapsto \mathbf{Z}''_1, \ldots, n_j \mapsto \mathbf{Z}''_j, \mathsf{below} \mapsto \mathbf{Z}''\} \\ \mathbf{Z}''_1, \ldots, \mathbf{Z}''_j, \mathbf{Z}'' \textit{ fresh} \qquad C' = k_1 \doteq k'_1 + \mathbf{Z}''_1 \wedge \cdots \wedge k_j \doteq k'_j + \mathbf{Z}''_j \wedge k \doteq k' + \mathbf{Z}'' \\ C'' = 0 < k_1 \leq \cdots \leq k_j \wedge 0 < k'_1 \leq \cdots \leq k'_j \wedge \mathbf{Z}''_1 \leq \cdots \leq \mathbf{Z}''_j \end{array}}{\Delta, \mathsf{access}(x, n_j, g) \textbf{ in } A \mid C \vdash \mathsf{access}(x, n_j, g') \textbf{ in } A' \Rightarrow \Delta, \mathsf{access}(x, n_j, g'') \textbf{ in } A' \mid C \wedge C' \wedge C''}$$

ATOM-MOVE-DOWN

$$\frac{\begin{array}{c} n_i \leq n_j \qquad A \leq A' \\ g = \{n_1 \mapsto k_1, \ldots, n_j \mapsto k_j, \mathsf{below} \mapsto k\} \qquad g' = \{n_1 \mapsto k'_1, \ldots, n_j \mapsto k'_j, \ldots, n_i \mapsto k'_i, \mathsf{below} \mapsto k'\} \\ g'' = \{n_1 \mapsto \mathbf{Z}''_1, \ldots, n_j \mapsto \mathbf{Z}''_j, \ldots, n_i \mapsto \mathbf{Z}''_i, \mathsf{below} \mapsto \mathbf{Z}''\} \qquad \mathbf{Z}''_1, \cdots, \mathbf{Z}''_i, \mathbf{Z}'' \textit{ fresh} \\ C' = k_1 \doteq k'_1 + \mathbf{Z}''_1 \wedge \cdots \wedge k_j \doteq k'_j + \mathbf{Z}''_j \wedge k \doteq 1 \wedge 1 \doteq k'_{j+1} + \mathbf{Z}''_{j+1} \wedge \cdots \wedge 1 \doteq k'_i + \mathbf{Z}''_i \wedge 1 \doteq k' + \mathbf{Z}'' \\ C'' = 0 < k_1 \leq \cdots \leq k_j \wedge 0 < k'_1 \leq \cdots \leq k'_i \wedge \mathbf{Z}''_1 \leq \cdots \leq \mathbf{Z}''_i \end{array}}{\Delta, \mathsf{access}(x, n_j, g) \textbf{ in } A \mid C \vdash \mathsf{access}(x, n_i, g') \textbf{ in } A' \Rightarrow \Delta, \mathsf{access}(x, n_i, g'') \textbf{ in } A' \mid C \wedge C' \wedge C''}$$

ATOM-MOVE-UP

$$\frac{\begin{array}{c} n_j \leq n_i \qquad A \leq A' \qquad g = \{n_1 \mapsto k_1, \ldots, n_i \mapsto k_i, \ldots, n_j \mapsto k_j, \mathsf{below} \mapsto k\} \\ g' = \{n_1 \mapsto k'_1, \ldots, n_i \mapsto k'_i, \mathsf{below} \mapsto k'\} \qquad g'' = \{n_1 \mapsto \mathbf{Z}''_1, \ldots, n_i \mapsto \mathbf{Z}''_i, \mathsf{below} \mapsto \mathbf{Z}''\} \\ \mathbf{Z}''_1, \cdots, \mathbf{Z}''_i, \mathbf{Z}'' \textit{ fresh} \qquad C' = k_1 \doteq k'_1 + \mathbf{Z}''_1 \wedge \cdots \wedge k_i \doteq k'_i + \mathbf{Z}''_i \wedge k_i \doteq 1 \\ C'' = 0 < k_1 \leq \cdots \leq k_j \wedge 0 < k'_1 \leq \cdots \leq k'_i \wedge \mathbf{Z}''_1 \leq \cdots \leq \mathbf{Z}''_i \end{array}}{\Delta, \mathsf{access}(x, n_j, g) \textbf{ in } A \mid C \vdash \mathsf{access}(x, n_i, g') \textbf{ in } A' \Rightarrow \Delta, \mathsf{access}(x, n_i, g'') \textbf{ in } A' \mid C \wedge C' \wedge C''}$$

ATOM-FAIL

$$\frac{\textit{no permission for } x \textit{ in } \Delta \textit{ that implies } A}{\Delta \mid C \vdash \mathsf{access}(x, n_i, g) \textbf{ in } A \Rightarrow \Delta \mid C \wedge \bot}$$

MERGE-MATCH

$$\frac{\begin{array}{c} \Delta, \mathsf{access}(x, n_j, g'') \mid C \wedge C' \wedge C'' \vdash P \Rightarrow \Psi \qquad g = \{n_1 \mapsto k_1, \ldots, n_j \mapsto k_j, \mathsf{below} \mapsto k\} \\ g' = \{n_1 \mapsto k'_1, \ldots, n_j \mapsto k'_j, \mathsf{below} \mapsto k'\} \qquad g'' = \{n_1 \mapsto \mathbf{Z}''_1, \ldots, n_j \mapsto \mathbf{Z}''_j, \mathsf{below} \mapsto \mathbf{Z}''\} \\ \mathbf{Z}''_1, \ldots, \mathbf{Z}''_j, \mathbf{Z}'' \textit{ fresh} \qquad C' = \mathbf{Z}''_1 \doteq k_1 + k'_1 \wedge \cdots \wedge \mathbf{Z}''_j \doteq k_j + k'_j \wedge \mathbf{Z}'' \doteq k + k' \\ C'' = k_1 \leq \cdots \leq k_j \wedge k'_1 \leq \cdots \leq k'_j \wedge \mathbf{Z}''_1 \leq \cdots \leq \mathbf{Z}''_j \end{array}}{\Delta, \mathsf{access}(x, n_j, g), \mathsf{access}(x, n_j, g') \mid C \vdash P \Rightarrow \Psi}$$

Figure 5.4: Proof rules for deriving constraints for atomic permission predicates

splits that precede a root "down" move. Moving the root down causes fractions for new nodes to be added to the fraction function, and those fractions are initially always 1 (see Figure 4.12).

But this well-formedness condition still permits node fractions in the fraction function to be 0, which is not legal for a "real" permission. Unlike the below fraction, which is 0 for pure permissions, node fractions can never be 0 according to our splitting rules (Figure 4.12). Intuitively, this is because the node fractions track how often permissions were split.

We force a permission to be real by adding a constraint $0 < k_1$, where $k_1$ is the fraction that the root node $n_1 = \mathsf{alive}$ is mapped to, which forces all node fractions to be strictly positive. Only permissions that are actually used for proving a predicate have to be real. Possibly-zero node fractions in a fraction function essentially represent slack permissions at the end of permission inference that can, but do not have to, be real (and often they are not because other constraints *force* their node fractions to be zero).

Let us see how these constraints are generated by our atomic proof rules.

- ATOM-ROOT-MATCH proves a permission for some root node $n_j$ if there is a permission with that same root node in the context. In this case we can add a "fresh" permission (a permission with a fresh fraction function) for the same root into the output context, which represents the "leftover" permission after splitting off the needed from the available permission. We introduce constraints that force the fractions in the needed and leftover permissions to sum up to the fractions of the given permission. We also introduce constraints for enforcing the well-formedness of our fraction functions. Notice that the leftover permission is not required to be "real".

- ATOM-MOVE-DOWN proves a permission with a smaller root $n_i$ than the one available in the context ($n_j$). We again add a fresh permission with the smaller root into the leftover context. We introduce constraints that relate the node fractions for nodes defined in the available permission in the same way as before, but in accordance with the original rule for moving a root down we require the existing "below" fraction to be equal to the literal 1. And since we are intuitively "filling in" fractions for the newly materialized nodes, the needed and leftover fractions for these new nodes have to sum up to 1 as well. Finally, we again introduce our well-formedness constraints and make the available and needed permissions real.

- ATOM-MOVE-UP does the opposite of the previous rule, although it introduces slightly stronger constraints. In this case, the available permission has a smaller root than the needed permission, so we have to move the root up. We again introduce a fresh permission with the new root and relate the fractions for nodes defined in all three permissions. In accordance with the original rule for moving up a root (Figure 4.12) we also require the available fraction for the new root to be equal to 1. Finally, we require permissions to be well-formed and the available and needed permissions to be real as before.

- ATOM-FAIL applies when none of the above rules for proving an atomic permission apply. This is either because there are no permissions for the specified variable, $x$, in the context, or because the available permissions have insufficient state information. In either case, this rule makes the constraints trivially unsatisfiable.

- MERGE-MATCH eagerly merges two permissions in the context with the same root into one and uses that new permission to prove the given predicate $P$ (first premise). This ensures that we always have the strongest possible permission available to prove the next one. Two permissions with the same

$$\otimes\,\mathrm{R}\quad \frac{\Delta \mid C \vdash P_1 \Rightarrow \Psi_1 \qquad \Psi_1 \vdash P_2 \Rightarrow \Psi_2}{\Delta \mid C \vdash P_1 \otimes P_2 \Rightarrow \Psi_2} \qquad \otimes\,\mathrm{L}\quad \frac{\Delta, P_1, P_2 \mid C \vdash P \Rightarrow \Psi}{\Delta, P_1 \otimes P_2 \mid C \vdash P \Rightarrow \Psi}$$

$$\oplus\,\mathrm{R}\quad \frac{\Delta \mid C \vdash P_1 \Rightarrow \Psi_1 \qquad \Delta \mid C \vdash P_2 \Rightarrow \Psi_2}{\Delta \mid C \vdash P_1 \oplus P_2 \Rightarrow \Psi_1 \,\&\, \Psi_2} \qquad \oplus\,\mathrm{L}\quad \frac{\Delta, P_1 \mid C \vdash P \Rightarrow \Psi_1 \qquad \Delta, P_2 \mid C \vdash P \Rightarrow \Psi_2}{\Delta, P_1 \oplus P_2 \mid C \vdash P \Rightarrow \Psi_1 \oplus \Psi_2}$$

$$\&\,\mathrm{R}\quad \frac{\Delta \mid C \vdash P_1 \Rightarrow \Psi_1 \qquad \Delta \mid C \vdash P_2 \Rightarrow \Psi_2}{\Delta \mid C \vdash P_1 \,\&\, P_2 \Rightarrow \Psi_1 \oplus \Psi_2} \qquad \&\,\mathrm{L}\quad \frac{\Delta, P_1 \mid C \vdash P \Rightarrow \Psi_1 \qquad \Delta, P_2 \mid C \vdash P \Rightarrow \Psi_2}{\Delta, P_1 \,\&\, P_2 \mid C \vdash P \Rightarrow \Psi_1 \,\&\, \Psi_2}$$

$$\forall\,\mathrm{R}\quad \frac{\Delta \mid C \vdash P \Rightarrow \Delta' \mid C'}{\Delta \mid C \vdash \forall z.P \Rightarrow \Delta' \mid C'} \qquad \forall\,\mathrm{L}\quad \frac{\Delta, [\mathbf{Z}/z]P' \mid C \vdash P \Rightarrow \Delta' \mid C'}{\Delta, \forall z.P' \mid C \vdash P \Rightarrow \Delta' \mid C'}$$

$$\exists\,\mathrm{R}\quad \frac{\Delta \mid C \vdash [\mathbf{Z}/z]P \Rightarrow \Delta' \mid C'}{\Delta \mid C \vdash \exists z.P \Rightarrow \Delta' \mid C'} \qquad \exists\,\mathrm{L}\quad \frac{\Delta, P' \mid C \vdash P \Rightarrow \Delta' \mid C'}{\Delta, \exists z.P' \mid C \vdash P \Rightarrow \Delta' \mid C'}$$

Figure 5.5: Proof rules for linear logic formulae

root occur when additional permissions are injected into the context from method post-conditions (cf. T-CALL in Figure 5.2).

The constraints generated for moving a root up are slightly stronger than the ones for moving a root down. This reflects the asymmetric nature of root moves: once a root was moved down, one cannot necessarily move it back up. (This is due to additional dimensions introduced in subclasses.) This has practical implications which we address in Section 6.4.2. But it is also, crucially, the reason for our lazy splitting scheme, which only moves a root down (and up) when we have to.

### Linear Connectives

Figure 5.5 summarizes the judgments for proving linear logic formulae other than atomic permissions. The rules for proving linear logic connectives ($\otimes$, $\&$, $\oplus$) are a straightforward adaptation of resource management for linear logic. In particular, constraints are simply threaded through the proof in the obvious way. Notice that we use our "choice" and "all" contexts for handling connectives other than $\otimes$ to avoid backtracking (see Section 5.1.1). Furthermore, we use conventional strategies for inferring quantifier instantiations. Notice that the rules for $\&$ and $\oplus$ as well as the rules for $\forall$ and $\exists$ are exactly dual to each other, as they should be.

$$\textsc{Ctx-}\,\&\quad \frac{\Psi_1 \vdash P \Rightarrow \Psi_1' \qquad \Psi_2 \vdash P \Rightarrow \Psi_2'}{\Psi_1 \,\&\, \Psi_2 \vdash P \Rightarrow \Psi_1' \,\&\, \Psi_2'} \qquad \textsc{Ctx-}\oplus\quad \frac{\Psi_1 \vdash P \Rightarrow \Psi_1' \qquad \Psi_2 \vdash P \Rightarrow \Psi_2'}{\Psi_1 \oplus \Psi_2 \vdash P \Rightarrow \Psi_1' \oplus \Psi_2'}$$

Figure 5.6: Context proof rules

#### Context Rules

Finally, the rules for breaking down choice and all contexts in order to invoke the linear logic proof judgment presented in the preceding sections are shown in Figure 5.6. The rules stipulate that predicates are proven separately for compound contexts and put back together into a new compound context afterwards.

#### Focusing

The rules as presented are still not fully deterministic: we can choose to break down a premise (and which premise) or to break down the conclusion. The order in which premises and conclusions are broken down is relevant, for instance, for proving $A \,\&\, B \vdash A \,\&\, B$ (if the premise is broken down before the conclusion then the proof will fail while breaking down the conclusion first will succeed).

Since there are only finitely many orders in which one could break down premises and conclusions (because they never get bigger), backtracking can be used to find an order that allows proving the needed conclusion. *Focusing* is the standard technique for making the choices for breaking down premises and conclusions deterministic (Andreoli, 1992). Since the formalization of focusing for linear logic is lengthy and standard we omit it here.

### 5.1.4 Soundness and Completeness

In the original presentation of permission splitting and merging we divided fractions (and fraction functions) in half and put them back together, using the intuition that $f = f/2 + f/2$. This was mostly done for notational convenience: it seems intuitive to think of splitting and merging as equating fractions $f = g + h$, but that would have required us to generate fresh fraction names everywhere, which we avoid in Chapter 4 for notational clarity.

This notational difference makes it hard to establish a direct correspondence between the two systems. It certainly appears that if there is a typing derivation in the original system then there is a derivation in the inference system with satisfiable constraints. The converse is less obvious. Nonetheless we believe that our inference algorithm is faithful to the original system and preserves safety, i.e., progress and preservation, *if* a derivation with satisfiable constraints can be found. Constraint satisfiability will be discussed in the following section.

## 5.2 Solving Constraints

The collection of constraints during typechecking allows us to effectively accumulate information about all the permissions needed in a piece of code. But a program should only typecheck if constraints are satisfiable.

$$\textsc{T-Meth}\ \frac{\mathit{this}, x_1, \ldots, x_n \mid P \vdash M : E \Rightarrow \Psi \qquad \models \Psi}{m(x_1, \ldots, x_n) : P \multimap E = M \in \Lambda}$$

$$\textsc{Sat-Atom}\ \frac{\models C}{\models \Delta \mid C} \qquad \textsc{Sat-\&-L}\ \frac{\models \Psi_1}{\models \Psi_1 \,\&\, \Psi_2} \qquad \textsc{Sat-\&-R}\ \frac{\models \Psi_2}{\models \Psi_1 \,\&\, \Psi_2} \qquad \textsc{Sat-}\oplus\ \frac{\models \Psi_1 \qquad \models \Psi_2}{\models \Psi_1 \oplus \Psi_2}$$

Figure 5.7: Well-defined methods

### 5.2.1 Checking Method Definitions

We could check satisfiability, written $\models C$, after typechecking every program expression. But there is no point since constraint satisfiability for the surrounding expression implies that the constraints for its sub-expressions are satisfied (because constraints are conjunctions that only ever grow in length). Therefore, we can *delay* constraint resolution to the outermost expression, which in our case are the method bodies. The rule for ensuring that a method body is well-defined is shown in Figure 5.7. The rules for context satisfiability require constraint satisfiability for atomic contexts and let us choose a component context in "choice" contexts, while both components must be satisfiable in "all" contexts.

### 5.2.2 Constraint Satisfiability

Constraints contain "unknown" variables, written $z$, and *logic variables* $\mathbf{Z}$. The quantifier proof rules of Figure 5.5 introduce both kinds of variables, while the atomic predicate rules (Figure 5.4) only introduce logic variables. Logic variables can be instantiated arbitrarily to satisfy constraints. Unknown variables, on the other hand, are assumed to have a fixed but unknown value. Nonetheless, we are allowed to make certain assumptions about them, which come from constraints for well-formedness and permissions being "real". In other words, assumptions about unknowns are formulae $0 < z_1 \leq \ldots \leq z_i$. Assumptions can never contain $\bot$.

In the interest of clarity we will explicitly quantify logic and unknown variables with existential and universal quantifiers, respectively. These quantifiers correspond to our intuition that logic variables can be instantiated with a particular value while unknowns can represent *any* fraction. Constraints therefore define the following logical formula:

$$\forall z_1, \ldots, z_n.(C_1 \implies (\exists \mathbf{Z}_1, \ldots, \mathbf{Z}_m.C_2)) \tag{5.1}$$

Here, $C_1$ is a conjunction of assumptions as described above and $C_2$ is an arbitrary constraint formula (see the constraint syntax in Figure 5.1). The $z_i$ are explicitly quantified unknown variables and the $\mathbf{Z}_j$ are explicitly quantified logic variables.

Constraints of this form (that are not trivially true or false) can be checked for satisfiability using Fourier-Motzkin elimination.[2] In a nutshell, Fourier-Motzkin allows eliminating a given variable from a conjunction of linear constraints by re-writing the input formula into an equivalent formula that does not include the

[2]Linear programming (Schrijver, 1998) can unfortunately not deal with alternatingly quantified variables directly.

eliminated variable (Schrijver, 1998). In particular, the input formula is satisfiable if and only if the new formula is. We can successively eliminate all variables from our constraints using Fourier-Motzkin, which will result in a final constraint formula that is equivalent to the input formula and contains only rational constants but no variables. Equality and inequality between rational constants (such as $0 < 1$ or $1 = 1$), however, is decidable.

In order to deal with the alternating quantification in (5.1) we use Fourier-Motzkin to first eliminate the existentially quantified variables, then rewrite the outer universal quantifier into an existential quantifier (which introduces negations), and finally use Fourier-Motzkin to eliminate the remaining and now existentially quantified variables. This proceeds as follows.

1. We eliminate the inner, existential quantifier (the $\mathbf{Z}_1, \ldots, \mathbf{Z}_m$ in equation (5.1) above). This yields a equivalent but only universally quantified formula:

$$\forall z_1, \ldots, z_n.(C_1 \implies C_2')$$

2. We replace all equality constraints of the form $k = k_1 + \ldots + k_n$ in $C_2'$ with the (equivalent) conjunction of relational constraints $(k \leq k_1 + \ldots + k_n) \wedge (k \geq k_1 + \ldots + k_n)$, yielding another equivalent formula:

$$\forall z_1, \ldots, z_n.(C_1 \implies C_2'')$$

3. Rewrite the formula with an existential quantifier and desugar the implication in the usual way:

$$\neg(\exists z_1, \ldots, z_n.\neg(\neg C_1 \vee C_2''))$$

   Which simplifies to

$$\neg(\exists z_1, \ldots, z_n.C_1 \wedge \neg C_2'')$$

4. Since $C_2'' = F_1 \wedge \ldots \wedge F_l$ is a conjunction of relational constraints, we can trivially negate each conjunct $F_i$, yielding new relational constraints $F_i'$. Applying DeMorgan's rules yields

$$\neg(\exists z_1, \ldots, z_n. \bigvee_{i \in 1,\ldots,l} C_1 \wedge F_i')$$

5. Finally, we push the existential quantifier inwards

$$\neg(\bigvee_{i \in 1,\ldots,l} \exists z_1, \ldots, z_n.C_1 \wedge F_i') \tag{5.2}$$

6. We can apply Fourier-Motzkin (or linear programming) to decide the satisfiability of each disjunct. The original constraints (5.1) are satisfiable if *and only if* all disjuncts in (5.2) are *un*satisfiable.

Thus, Fourier-Motzkin elimination can be used to check our fraction constraint formulae (5.1) for satisfiability. Nipkow (2008) discusses more sophisticated algorithms for eliminating linear quantifiers, which are applicable to our constraint formulae. We leave the evaluation of these algorithms to future work.

As it turns out, the order in which variables are eliminated in the first step affects the length of the resulting formula. Shorter formulae result from eliminating variables from right to left in relational constraints $\mathbf{Z}_1 \leq \ldots \leq \mathbf{Z}_n$ because that minimizes the number of other variables that the currently eliminated variable is related to.

# Chapter 6

# Plural: Java Tooling

Our prototype tool, Plural[1] ("Permissions Let Us Reason about ALiasing"), is a plug-in to the Eclipse IDE that embodies the permission inference approach presented in the preceding chapter as a static dataflow analysis for Java. In the remainder of this chapter we show example annotations and explain how permissions are tracked and API implementations are verified. Then we discuss tool features we found useful in practice and for dealing with a real programming language. Notice that Plural assumes the analyzed program to be properly synchronized; possible thread interleavings over the analyzed program are not considered.

Nels Beckman helped with implementing the Plural tool and in particular contributed to API implementation checking (Section 6.3.7) and tracking concrete predicates (Section 6.4.6). He also implemented an extension to Plural, called NIMBY, that checks protocol compliance of concurrent software using transactional memory for synchronization (Beckman et al., 2008).

## 6.1 Developer Annotations

Developers use Java 5 annotations to specify method pre- and post-conditions with access permissions. Figure 6.1 shows a simplified `ResultSet` protocol with Plural's annotations (recall Figure 1.1 and the discussion of JDBC protocols in Chapter 1). Annotations on methods (parameters) specify *borrowed* permissions for the receiver (resp. the annotated parameter). Borrowed permissions are returned to the caller when the method returns. The attribute "guarantee" specifies a state that cannot be left while the method executes. For example, `next` advances to the next row in the query result, guaranteeing the result set to remain *open*. Conversely, a required (ensured) state only has to hold when the method is called (returns). For instance, only after calling `getInt` is it legal to call `wasNull`.

Annotations are therefore the main way in which developers interact with Plural. Major annotations include the following.

- `@Unique`, `@Full`, `@Share`, `@Imm`, and `@Pure` declare unique, full, share, immutable, and pure permissions, respectively, for the annotated method parameter or, if they are placed on a method, the method's receiver[2]. They define the following attributes:

[1] `http://code.google.com/p/pluralism/`

[2] Receiver annotations could be moved to the "receiver" annotation position when JSR-308 is adopted. JSR-308 proposes

```
@Param(name = "stmt", releasedFrom("open"))
public interface ResultSet {
    @Full(guarantee = "open")
    @TrueIndicates("unread")
    @FalseIndicates("end")
    boolean next();

    @Full(guarantee = "valid", ensures = "read")
    int getInt(int column);

    @Pure(guarantee = "valid", requires = "read")
    boolean wasNull();

    @Full(ensures = "closed")
    @Release("stmt")
    void close();
}
```

Figure 6.1: Simplified `ResultSet` specification in Plural (using the typestates shown in Figure 1.1).

  - `guarantee` declares a state guarantee, also known as the permission's root node. This is also the default attribute.
  - `requires` and `ensures` declare required and ensured states, respectively.
  - `returned` can be set to `false` to declare *consumed* permissions; otherwise, the permission is *borrowed* (Section 6.4.2).
  - `use` specifies whether the permission is used for calling other methods (default), accessing fields, or both. The first two options distinguish virtual and frame permissions (see Section 4.1.5); the last one was added for pragmatic concerns and will be further explained in Section 6.4.7.

- `@ResultUnique`, etc., declare permissions for method return values in the same way as described above. These annotations syntactically distinguish receiver from return value permissions.
- `@TrueIndicates` and `@FalseIndicates` declare a method to be a dynamic state test with Boolean return type, where the return value *true* (*false*, respectively) indicates the receiver (when placed on a method) or parameter to be in the given state (Section 6.3.6).
- `@Perm` can be used to define complex method pre- and post-conditions with the textual attributes `requires` and `ensures`. Individual permissions are expressed with the syntax we use in the formal system (see Chapter 4); `*`, `&`, and `+` are used to represent linear connectives; and concrete predicates such as the truth of a Boolean variable can also be asserted. The content of this annotation is combined with any permission annotations (explained above) declared for the same method.
- `@Cases` is used to declare multiple cases for a single method, using `@Perm` for each case.

allowing annotations in additional syntactic positions including for method receivers, array contents, and type parameters in a future version of Java. Plural could benefit from all of these.

- `@State` defines a state invariant using the same textual syntax as `@Perm`. This annotation is aggregated with `@ClassStates`.
- `@Param` declares a permission parameter (Section 6.4.5); `@Capture` captures and `@Release` releases such a parameter (see Section 7.1.1 for examples).
- `@Lend` annotates a parameter to "lend" the result of a method, temporarily invalidating that parameter until the result is no longer used (example in Figure 7.5).
- `@IsResult` can be used to declare that a method parameter is returned as the method result.

Plural's annotations for license-technical reasons are homed in a separate open-source project[3] which contains documentation for these and other annotations.

## 6.2 Background

This section provides background on the Eclipse IDE and the Crystal static analysis framework, itself a Eclipse plug-in. The Plural tool relies on these.

### 6.2.1 Eclipse

Eclipse[4] is a popular open-source integrated development environment (IDE) for Java.[5] It provides a rich infrastructure for developing plug-ins that augment the standard Eclipse features. Two capabilities of Eclipse are essential for Plural to perform its task: the Java AST and the "problems view".

Eclipse provides a parser, typechecker, and abstract syntax tree (AST) for Java source files. This alleviates Crystal and Plural from having to parse, typecheck, and represent Java code, which is a complicated undertaking by itself. Since it is well–tested, the AST and typing information provided by Eclipse should be (mostly) reliable. We did find several bugs in how Eclipse resolves Java 5 annotations, all of which were fixed with Eclipse release 3.4.1.

Plural uses the "problems view" to communicate possible protocol violations to the developer. This allows Plural to conveniently associate problems with a location in the analyzed Java source file (usually a method or constructor call site) and provide a textual description of the problem.

### 6.2.2 Crystal

Crystal[6] is a static analysis framework co-developed by the author. Crystal is itself an Eclipse plug-in that in particular takes advantage of Eclipse's AST. Crystal provides an implementation of a *worklist* algorithm for performing dataflow analyses (Nielson et al., 1999) on Java source files. It also provides an abstraction of Java source code as *3-address code*.

3-address code represents nested Java expressions as sequences of atomic instructions that assign to temporary variables and only use variables as operands. Thanks to aggressive desugaring, the many different

[3] `http://code.google.com/p/plaidannotations/`

[4] `http://www.eclipse.org/`

[5] Eclipse offers support for editing non-Java artifacts, but these capabilities are not important for this discussion.

[6] `http://code.google.com/p/crystalsaf/`

Java expressions can be represented with approximately 20 kinds of instructions. This reduction simplifies writing source code analyses since there are fewer cases to consider.

Flow analyses in Crystal are defined with a *transfer function*. A transfer function takes incoming analysis information and a "node", i.e., an operation in the control flow of a method, and computes the resulting outgoing analysis information. Transfer functions are analysis-specific, and Plural implements several transfer functions to perform its task. Crystal's worklist algorithm calls a given transfer function to transfer over the nodes in the control flow graph of the analyzed method. Transfer functions can choose to transfer directly over Eclipse Java AST nodes or on 3-address code instructions; Plural is based on the latter.

Crystal's dataflow analysis infrastructure provides what we call *branch sensitivity*, which was crucial for supporting dynamic state tests in Plural (cf. Section 6.3.6). A *branch* is a point in the control flow with multiple successors that control can transfer to at runtime. Common examples for branches are Boolean tests and operations that may result in exceptions. Crystal "labels" branches with the conditions under which they will be taken. In particular, it defines labels for "true" and "false" outcomes of Boolean tests and for exceptions.

When Crystal's worklist algorithm invokes a transfer function to transfer over a branch it signals the labels on the branches to the transfer function. The transfer function then has to opportunity to provide outgoing analysis information *separately* for each label. For example, it can provide separate information for the "true" and "false" branches of a Boolean test. Crystal will propagate information associated with a label only on branches with that label, which will ultimately allow analyzing operations following Boolean tests *assuming* a certain test outcome.

Analysis information is specific to a transfer function and is therefore defined by analysis writers; Crystal's worklist algorithm is parameterized by the specific analysis information being tracked by a given transfer function. Two operations must be implemented for *comparing* and *joining* analysis information at control flow merge points. Plural's implementation of these operations is discussed below.

Branch sensitivity can be used to implement *path-sensitive* analyses: at control flow merge points, analysis writers can choose to keep analysis information computed for the different incoming paths separate by using a suitable *join* implementation. However, Plural does *not* implement such a *join* and instead approximates analysis information from different paths at control flow merge points.

## 6.3 Flow Analysis for Local Permission Inference

Like any dataflow analysis, Plural tracks its analysis information in a lattice. Intuitively, the information Plural tracks are the permissions available for each object. In Chapter 5 we discussed how permissions can be inferred automatically for a program using constraints, and how composite contexts allow reasoning about linear logic $\oplus$ and $\&$ connectives. Plural tracks constraints as part of its flow analysis information and includes support for composite contexts, which will be discussed below. Plural's transfer functions essentially implement the permission inference rules described in Chapter 5 and will therefore not be discussed in detail here.

### 6.3.1 Annotations Make Analysis Modular

Figure 6.2 shows a simple client method that retrieves an integer value from the first column in the first row of the given result set. Plural can be used to check that this code respects the protocol declared for the

```
public static Integer getFirstInt(@Full(guarantee = "open") ResultSet rs) {
    Integer result = null;
    if(rs.next()) {
        result = rs.getInt(1);
        if(rs.wasNull())
            result = null;
        return result;
    }
    else {
        return rs.getInt(1); // ERROR: rs in "end" instead of "valid"
    }
}
```

Figure 6.2: Simple `ResultSet` client with error in *else* branch that is detected by Plural.

`ResultSet` interface in Figure 6.1.

Our goal is to avoid annotations inside method bodies completely: based on the declared protocols, Plural infers how permissions flow through method bodies. Using the lattice comparison and join operations described in the next subsection, Plural can automatically infer loop invariants.

However, Plural does require additional annotations on method parameters that have a declared protocol, such as the `ResultSet` parameter in Figure 6.2. We annotate client code with the same annotations that we use for declaring API protocols. While protocol annotations on the API itself (e.g., Figure 6.1) can conceivably be provided by the API designer and amortize over the many uses of that API, the annotation shown in Figure 6.2 is specific to this client program. In Section 7.4 we discuss the overhead of providing these additional annotations for two open-source codebases.

Annotations make the analysis modular: Plural checks each method separately, temporarily trusting annotations on called methods and checking their bodies separately. For checking a given method or constructor, Plural assumes the permissions required by the method's annotations, i.e., it assumes the declared pre-condition. At each call site, Plural makes sure that permissions required for the call are available, splits them off (these permissions are "consumed" by the called method or constructor), and merges permissions ensured by the called method or constructor back into the current context. Notice that most methods "borrow" permissions (cf. Figure 6.1), which means that they are both required and ensured. At method exit points, Plural checks that permissions ensured by its annotations are available, i.e., it checks the declared post-condition.

Thus, permissions are handled by Plural akin to conventional Java typing information: Permissions are provided with annotations on method parameters and then tracked automatically through the method body, like conventional types for method parameters. Unlike with Java types, local variables do not need to be annotated with permissions; instead, their permissions are inferred by Plural. Permission annotations can be seen as augmenting method signatures. They do not affect the conventional Java execution semantics; instead, they provide a static guarantee of protocol compliance without any runtime overhead.

### 6.3.2 Tuple Lattice

At the heart of Plural's analysis is a tuple lattice for tracking permissions and constraints for individual objects. A tuple lattice in general tracks separate analysis information for each object and is compared and

joined pointwise. The information tracked for each object is a pair: the set of permissions currently available for the object (at most one for each orthogonal dimension) together with the constraints collected for these permissions.[7]

Such a tuple conceptually corresponds to an atomic context as discussed in Chapter 5. However, it is a simplification because atomic contexts can in general contain linear logic formulae, whereas Plural tuples contain atomic permissions only. Formulae, e.g., from a method post-condition, are broken down into their constituent permissions before merging them into a given tuple. This has no consequences on precision in the case of multiplicative conjunctions, $P_1 \otimes P_2$, where permissions $P_1$ and $P_2$ can be separately merged into the tuple. But this strategy can result is a loss of precision when inserting $\oplus$ and $\&$ formulae. However, we *never* explicitly used such formulae in our case studies, so this issue has had no practical consequences.[8]

**Representing individual permissions.** Permissions in Plural are uniformly represented with their root node, fraction function, state information, and a flag distinguishing read-only permissions (pure and immutable). As before, the fraction function is a mapping from nodes in the state hierarchy to fractions together with a "below" fraction. Permissions declared in annotations will contain literal fractions such as 1 and 0, while permissions generated by permission inference rules will usually contain fraction variables only.

### 6.3.3 Comparing and Joining Permission Tuples

When comparing sets of permissions for the same object (as mentioned, tuples are compared and joined pointwise), Plural pairs up permissions in the two sets with the same root node (or related root nodes). It then compares these permissions based on several criteria including the following:

1. Additional permissions make that permission set more precise. If both sets contain permissions for nodes not contained in the other set then the sets are incomparable.

2. When comparing permissions with the same root it compares fractions for the same node pairwise and treats

   - pairwise equal fractions in the fraction function as equal,
   - logic variable ($\mathbf{Z}$) or literal fractions (such as 1) in place of unknown fractions ($z$) from existential quantifiers in the other permission as more precise[9], and
   - all other fractions as incomparable.

3. if there are permissions for subnodes of a node in one set for which the other set contains a permission for that node, Plural attempts to align these permissions by moving roots up or down and then use the

[7] In Chapter 5 we used one set of constraints, but Plural keeps constraints for different objects separate to simplify satisfiability checking and error reporting.

[8] Dynamic state tests and method cases (Sections 6.3.6 and 6.4.3) are treated specially and do not suffer from these imprecisions.

[9] Care must be taken not to treat unknown fractions from the checked method's pre-condition as less precise than other fractions, as this would compromise our treatment of borrowed permissions (Section 6.4.2). The difference is that unknowns from pre-conditions result from unpacking universal quantifiers, while the unknowns introduced in joining permissions result from existential quantifiers, which can be alpha-converted.

above rules to compare the new sets; if root movements are not possible then the permission sets are considered incomparable.

The same rules guide the process of joining permission sets: excess permissions are dropped, incomparable pairs of permissions are approximated (described below), and permissions with misaligned nodes are either aligned (through root movement) or dropped (if alignment is not possible).

A pair of incomparable permissions is approximated by generating a new permission with fresh *unknown* fractions (in contrast to the fresh variable fractions introduced by the permission inference). Because these fresh unknown fractions have an unknown value they are deemed less precise than variable and literal fractions, as mentioned in the description of Plural's comparison rules.

Notice that we do *not* attempt to determine whether a fraction is smaller than another given a set of constraints, although this would allow comparison of permissions incomparable in the above rules (larger fractions are more precise). We have not found this to be a problem in practice, but it appears that such a refinement of our lattice comparison algorithm would be possible.

Our strategy of introducing unknown fractions to join incomparable permissions marks a difference to previous work on inferring fractions (Terauchi and Aiken, 2008; Terauchi, 2008). Previously, control flow merges introduced new fraction variables that were forced to be smaller than the incoming fractions. Function signatures were also inferred with fraction variables. Constraints could then be collected by scanning the entire program once, even in the presence of conditionals and loops, allowing the process of constraint collection to be asymptotically linear in the size of the program and avoiding the complications of deciding constraints with unknown variables (see Section 5.2.2).

Conversely, we collect constraints with a flow analysis, which is polynomial in the worst case (Nielson et al., 1999, Chapter 6). We could nonetheless use the previous approach for joining permissions for control flow merges resulting from *if* statements, but it would not terminate for loops. An advantage of our approach is that permissions can be consumed in loops, which was not previously possible (see Terauchi, 2008, Section 5). While most methods do not consume permissions, some do in the APIs we investigated (see Chapter 7), usually those methods related to creating new objects based on existing ones. We are also able to use universal quantification in function signatures (currently provided with explicit annotations), which may be more flexible than inferring concrete fractions for signatures, as was done previously.

**Tuple lattice has finite height.** As with any dataflow analysis, the question arises whether ours is guaranteed to terminate, which, because of the iterative nature of the algorithm, can be reduced to showing that the lattice in use has finite height. We informally argue that this is the case.

- Tuple lattices have finite height (for a finite set of objects) if the information tracked for an individual object forms a lattice of finite height. In our case, this information is the permission set tracked for an object.

- Our permission sets are conceptually akin to a set intersection lattice, where the empty set represents the least amount of information (notice that we drop permissions present in one but not the other set when joining sets). The size of our sets is finite because there can be only at most one permission rooted in each node of the tracked object's state space in it, and that state space, being user-defined, is finite. (Recall that permission inference merges permissions with the same root node, see Chapter 5.)

- When joining individual permissions we either (a) preserve the fractions present in these permissions (if they are equal) or (b) approximate them with an unknown variable. Already approximated fractions will not be approximated again because the unknowns that were introduced are less precise than any other fraction (including other unknowns), terminating the process of approximation of individual permissions after one iteration.

**Future work: interprocedural analysis.** We believe that the approach for comparing and joining permission tuples described above could be used in an interprocedural analysis as well. In particular, methods being called from multiple call sites will cause the tuples available at the different call sites to be joined, possibly introducing unknown variables, which directly represent the universally quantified fractions we typically find in developer-defined method pre-conditions. Such an approach appears to have similarities to *let-polymorphism* in Hindley-Milner type inference (Pierce, 2002).

This would allow automatically analyzing larger program modules, such as a set of related classes or an entire program. However, constraint sets would possibly grow to very large sizes, which has been a problem in previous work when analyzing a whole program at once (Terauchi, 2008).

### 6.3.4 Composing Tuples

The tuple lattice discussed above induces a lattice of composite tuples that correspond to the composite *choice* and *all* contexts from Section 5.1. Formally, we define the elements $A$ of this lattice as follows:

$$\begin{array}{rcll} A, B, C & ::= & T & \textit{atomic tuple} \\ & | & A \,\&\, B & \textit{choice element} \\ & | & A \oplus B & \textit{all element} \\ & | & \top & \textit{top} \\ & | & \bot & \textit{bottom} \end{array}$$

Comparing the elements of this lattice is straightforward, given the comparison $T_1 \sqsubseteq T_2$ for individual tuples we described in Section 6.3.3.

$$\frac{A \sqsubseteq C}{A \,\&\, B \sqsubseteq C} \qquad \frac{B \sqsubseteq C}{A \,\&\, B \sqsubseteq C} \qquad \frac{A \sqsubseteq B \qquad A \sqsubseteq C}{A \sqsubseteq B \,\&\, C} \qquad \frac{}{A \sqsubseteq \top}$$

$$\frac{A \sqsubseteq C \qquad B \sqsubseteq C}{A \oplus B \sqsubseteq C} \qquad \frac{A \sqsubseteq B}{A \sqsubseteq B \oplus C} \qquad \frac{A \sqsubseteq C}{A \sqsubseteq B \oplus C} \qquad \frac{}{\bot \sqsubseteq A}$$

These correspond exactly to the subtyping rules for union and intersection types, where $A \,\&\, B$ corresponds to intersections and $A \oplus B$ corresponds to unions (Dunfield and Pfenning, 2004).

This lattice allows reasoning about linear logic predicates that include internal ($\&$) and external choice ($\oplus$) operators. Unfortunately, however, the lattice has infinite height. We can finitize it by limiting the nesting depth of tuples. For example, we can restrict ourselves to only allow atomic tuples in $\&$-elements and only $\&$-elements in $\oplus$-elements, similar to a disjunctive normal form. Such a restriction is not complete with respect to the infinite lattice because of our limitation to a fragment of linear logic. In particular,

the problem is that in the constructive fragment we are using (which does not include negation), certain De Morgan rules only work in one direction but not the other. For example $(A \,\&\, B) \otimes C \vdash (A \otimes C) \,\&\, (B \otimes C)$, but $(A \otimes C) \,\&\, (B \otimes C) \nvdash (A \,\&\, B) \otimes C$.

The implementation of Plural is currently restricted to only use $\&$-elements. This simplification was possible because we only used conjunctions in our case studies, never internal or external choices. Furthermore, we suspect that the added benefit of $\oplus$-elements would be small because conventional joining of lattice elements achieves something very similar to $\oplus$-elements. Joining approximates two elements into one that contains as much information from the two original elements as possible. The resulting lattice element must then be able to satisfy the subsequent predicates. With $\oplus$-elements, Plural still has to make sure that *both* alternatives satisfy all subsequent predicates. $\&$-elements, on the other hand, allow delaying choices, which is in particular useful in the context of API implementation checking and for handling method cases.

### 6.3.5 Local Must Alias Analysis

Plural uses a simple local alias analysis to identify local variables that are known to point to the same object. The local alias analysis tracks a set of "locations" for each local variable. These locations are effectively Plural's static approximation of the different objects that a local variable may point to at run time. The tuple lattice described above therefore associates permission sets and constraints with these locations.

Plural generates fresh locations for references returned from method calls. These references may at runtime point to the same object as another local variable. In other words, local variables may alias even though our local alias analysis does not indicate this possibility. The use of permissions prevents problems in this case because Plural will associate different sets of permissions with the two locations which correspond to the same runtime object.

The purpose of Plural's local alias analysis is therefore *not* to soundly approximate all possible runtime aliasing, but to identify local variables that must alias and can therefore access the same permissions. The local alias analysis in particular prevents us from having to split permissions when a local is assigned to another local with a *copy* instruction, such as `x = y`, which avoids developer-provided annotations in method bodies.

### 6.3.6 Dynamic State Tests

APIs often include methods whose return value indicates the current state of an object, which we call *dynamic state tests*. For example, `next` in Figure 6.1 is specified to return `true` if the cursor was advanced to a *valid* row and `false` otherwise. Sensitivity to such tests is part of the type system underlying Plural (Chapter 4); this section discusses how Plural automatically reasons about dynamic state tests using Crystal's support for branch sensitivity.

In order to take such tests into account, Plural performs a *branch-sensitive* flow analysis: if the code tests the state of an object, for instance with an `if` statement, then the analysis updates the state of the object being tested *according to the test's result*. For example, Plural updates the result set's state to *unread* at the beginning of the outer *if* branch in Figure 6.2. Likewise, Plural updates the result set's state to *end* in the *else* branch and, consequently, signals an error on the call to `getInt`. Section 6.4.6 explains how these lattice manipulations are implemented internally.

Notice that this approach does not make Plural path-sensitive: analysis information is still joined at control-flow merge points. Thus, at the end of Figure 6.2, Plural no longer remembers that there was a path

through the method on which the result set was *valid*. We believe that Plural could be extended to retain this information, but then we would have to deal with the usual complications of path sensitivity, i.e., large or infinite numbers of paths even through small methods.

When checking the implementation of a state test method, Plural checks at every method exit that, assuming `true` (or `false`) is returned, the receiver is in the state indicated by `true` (resp. `false`). This approach can be extended to other return types, although reasoning about predicates such as integer ranges may require using a theorem prover (Bierhoff and Aldrich, 2008; Leino and Müller, 2009).

### 6.3.7 API Implementation Checking

Our approach not only allows checking whether a client of an API follows the protocol required by that API, it can also check that code implementing an API is safe when used with its declared protocol. The key abstraction for this are *state invariants*, which we adapted from Fugue (DeLine and Fähndrich, 2004b, see also Chapter 3.2). A state invariant associates a typestate of a class with a predicate over the fields of that class. In our approach, this predicate usually consists of access permissions for fields. An example can be found in Figure 7.7.

Whenever a receiver field is used, Plural *unpacks* a permission to the surrounding object to gain access to its state invariants (Bierhoff and Aldrich, 2007b). Essentially, unpacking means replacing the receiver permission with permissions for the receiver's fields as specified in state invariants. Before method calls, and before the analyzed method returns, Plural *packs* the receiver, possibly to a different state, by splitting off that state's invariant from the available field permissions (Bierhoff and Aldrich, 2007b). Plural sometimes has to "guess" what state to pack to by trying all possibilities, which our "choice" contexts allow us to do.

### 6.3.8 Error Reporting

Plural reports possible protocol violations at the point in the program where errors are detected. In particular, if a method pre-condition cannot be satisfied at a call site then Plural will issue a warning at that call site, explaining the nature of the violation. Plural finds violations by querying the results of the dataflow analysis and checking at each method call site whether the pre-condition is satisfiable. Note that the actual error may precede the point in the method where a protocol violation occurs, for instance because an earlier method call may set up a situation where a later pre-condition becomes unsatisfiable.

## 6.4 Extensions

This section describes features implemented in Plural that are *not* captured in the theory underlying the tool but which we found useful in practice (see Chapter 7).

### 6.4.1 Immutable Permissions

While not part of the theory for permission reasoning presented in the preceding sections, we chose to include immutable permissions in Plural because of their apparent relevance notably for specifying collections and iterators (Section 3.1). We track immutable exactly like share permissions; the only difference is that we will not allow assignments to fields when unpacking an immutable permission. Therefore, the simple "read-only" flag mentioned in Section 6.3.2 is sufficient for distinguishing immutable from share permissions in

the lattice. However, we must avoid immutable and share permissions from co-existing; in particular, we must never split off an immutable from a share permission or vice versa. Instead, we must re-assemble a full (or stronger) permission before switching from share to immutable or the other way around.

In terms of constraints this means that splitting a share from a read-only or a immutable from a mutable permission $p$, we generate an additional constraint that forces $p$ to be full, i.e., we force its "below" fraction to be 1. This is because a full permission is required to transition between objects immutable and objects being share.

### 6.4.2 Borrowing

Over and over again we encountered methods that *borrow* permissions, meaning they return the same permission that was passed in when they were invoked, although possibly with new state information.[10] Technically, our type system cannot express borrowing because it only quantifies over fractions inside a method pre- or post-condition, but over not both. Borrowed permissions are essentially permissions whose fractions are quantified over across both the method pre- and post-condition.[11]

Support for borrowing has two important implications:

- **Precision.** When a method call requires to move the root of a permission available at the call site down, and this permission is borrowed, then it is *always* safe to move the root back to where it was before the call. Without support for borrowing, the following code would flag an error, while Plural (correctly) does not flag an error at the call to `remove`.

  ```
  public static <T> T removeNext(@Full(requires = "available") Iterator<T> it) {
      T result = it.next();
      it.remove(); // annotated with @Full(guarantee = "current")
      return result;
  }
  ```

  The reason is that the given full permission is strong enough to move the root from alive to current, but not back up (moving a root up requires a unique permission). With borrowing, we know that we can move the root back to where it was.

- **Performance.** Support for borrowing also seems to improve performance: it cut the time Plural took to run over its regression test suite in half (from 70 to about 35 seconds on the author's machine). This is because we do not have to keep constraints introduced for borrowed permissions beyond the current call site and can instead revert to the permission that was previously in the lattice, although possibly with changed state information. Thus, borrowed permissions prevent constraints from accumulating over consecutive call sites, which simplifies determining whether constraints are satisfiable in the end. (This optimization is only sound if constraint violations are detected immediately, which Plural does.)

### 6.4.3 Method Cases

The idea of method cases goes back to behavioral specification methods, e.g., in the JML (Leavens et al., 1999). Method cases amount to specifying the same method with multiple pre-/post-condition pairs, allowing methods to behave differently in different situations. We early on recognized their relevance for

[10] The other common case is that permissions are *consumed*, i.e. not returned to the caller of a method.

[11] Such quantifiers could be added to the type system with additional notational overhead in the rules for checking method calls.

specifying API protocols (Bierhoff and Aldrich, 2005; Bierhoff, 2006), but we are not aware of any other protocol checking approaches that support method cases.

In terms of linear logic, method cases can be thought of as a choice between implications. For example, $(P_1 \multimap P_2) \,\&\, (P_3 \multimap P_4)$ encodes a method with two cases $P_1 \multimap P_2$ and $P_3 \multimap P_4$.

In order to support method cases, Plural tracks the possible permissions after a call to a method with cases using "choice" lattice elements. Checking the implementation of a method with multiple cases amounts to checking the method separately for each case.

### 6.4.4 Marker States

Plural can treat states special that are fixed throughout the object lifetime. We call these *marker states*, which are reminiscent of (flow-insensitive) type qualifiers (Foster et al., 2002) and type refinements (Dunfield and Pfenning, 2004). For example, result sets can be marked as *updatable* or *readonly* (see Section 7.1.1), and they cannot switch from one to the other once created. Knowledge about an object being in a marker state, once gained, cannot be lost, which can simplify checking API clients. Otherwise, marker states are technically no different from regular states. But they may also be informative to a human reading a specification: marker states indicate object properties that are fixed at construction time, thereby directly refining conventional Java types with additional, flow-insensitive information that does not change throughout the object's lifetime.

Notably, marker states can capture the well-known distinction between "modifiable" and "unmodifiable" collections as they are defined in the Java Collections Framework. We can also use marker states for distinguishing "readonly" and "modifying" iterators over mutable collections (Section 7.1.2).

### 6.4.5 Dependent Objects

Another feature of many APIs is that objects can become invalid if other, related objects are manipulated in certain ways. For example, SQL query results become invalid when the originating database connection is closed. (A similar problem, called concurrent modification, exists with iterators (Bierhoff, 2006).) There are no automated modular protocol checkers that we know of that can handle these protocols, although recent whole-program protocol checking approaches can (Bodden et al., 2008; Naeem and Lhoták, 2008).

**Permission Parameters.** Our solution is to "capture" a permission in the dependent object (the result set in the example) which prevents the problematic operation (closing the connection in the example) from happening. The dependent object has to be invalidated before "releasing" the captured permission and re-enabling the previously forbidden operation. Captured permissions are declared in Plural with a `@Param` annotation, and `@Release` explicitly releases permissions, as in `close` (Figure 6.1).

**Garbage Collection.** Others have modeled dependent objects with linear implications (Boyland and Retert, 2005; Krishnaswami, 2006; Haack and Hurlin, 2008) but it is unclear how well those approaches can be automated. Our solution is to use a live variable analysis to detect *dead objects*, i.e., dead references to objects with `unique` permissions, and release any captured permissions from these dead objects.[12]

[12] We could check that these objects are deleted (in C or C++) or mark them as available for garbage collection (in Java or C#), but we are not exploring this optimization possibility here.

### 6.4.6 Concrete Predicates

With concrete predicates we mean information about *values* such as primitive Booleans as well as the null-ness of references. Tracking the former is necessary for properly handling dynamic state tests, while the latter is useful in implementation predicates as well as for avoiding null dereference errors. At the end of this section we discuss how integers could be tracked.

Every *test* in a program, for example in an *if* statement or loop header, ascertains the truth or falsehood of some (temporary) variable at run time. This information often *implies* or *indicates* some other fact. For example, truth of the JDBC `ResultSet`'s `next` method's return value indicates that the method call's receiver is in the valid state (see Figure 6.1).

Plural's lattice pairs the tuples described above with a "dynamic state logic" that maintains (a) knowledge about concrete predicates (truth, falsehood, and nullness) and (b) a list of known implications. The latter mirror dynamic state test annotations such as `@TrueIndicates` with implications from the indicating Boolean value to the indicated state. Plural leverages Crystal's support for branch sensitivity to pass truth and falsehood of the tested variable down the respective branches. It eagerly eliminates applicable implications when knowledge about the value of a Boolean variable becomes available.

Plural handles the fact that a reference is "null" or "non-null" similar to a Boolean fact about a variable. If null-ness is tested at run time, Plural produces an implication from a Boolean to a null-ness fact as described above.

**Future Work: Integers.** Facts about integer variables, such as an integer being positive, could conceivably be tracked by Plural as well. Unlike with Boolean and null-ness facts, however, integers are notoriously difficult to reason about, and facts about them are hard to track precisely and efficiently. In the spirit of Plural, one could use sufficiently precise and efficient lattices to track integer ranges, as in recent work by Ferrara et al. (2008). Alternatively, one could employ a theorem prover, as is common in program verification (Flanagan et al., 2002), which would increase reasoning power but also overhead and might decrease predictability.

**Discussion.** Typestates are a promising framework for tracking concrete predicates more precisely. What we mean is that the tracking of concrete predicates discussed above is a well-studied area, often even commonly tracked with dataflow analyses based on simple lattices. But we found that concrete predicates often depend on an object's typestate: for instance, a field may only be non-null in a certain state and otherwise be null. We therefore believe that wrapping a typestate framework, such as Plural's, around these well-known techniques for tracking concrete predicates would allow tracking them in the appropriate context (e.g., assuming a particular typestate) and yield higher precision. In particular, considerable research effort has gone into reasoning about object initialization (Fähndrich and Xia, 2007; Qi and Myers, 2009), which we believe can be largely encoded with typestates.

At the same time, concrete predicates are crucial for reasoning about dynamic state tests, thereby helping our typestate analysis. While Plural only supports state test methods of Boolean return type, we have seen integers and null-ness as well as regular or exceptional control flow indicate states.

### 6.4.7 Permissions Allowing Dispatch *And* Field Access

In Section 4.1.5 we saw how a distinction into *frame* and *virtual* permissions allows using permissions smoothly with inheritance. Roughly, frame permissions allow field access (only), and virtual permissions allow calling other methods using dynamic dispatch (only). For a caller, this distinction is irrelevant: virtual permissions have to be provided by callers in either case, since virtual permissions are coerced into frame permissions as needed.

In our case studies we encountered code which accessed fields and called other methods in the body of the same method. In particular, the restriction to one or the other accounted for 42 out of 77 false positives in the parts of PMD we considered for one of our case studies (see Section 7.3.2).

Sometimes, we can use separate permissions to specify these methods, for instance a pure frame permission for reading fields and a full virtual permission for calling another method that required it. But at other times, a full frame permission will be needed to modify fields or the content of fields, and a full virtual permission is required by called methods. Such a method pre-condition, however, can never be satisfied because callers would have to provide two full permissions for the same object, which is by design impossible.

We remedied this problem by introducing permission annotations allowing *both* field access and dynamic dispatch. Plural checks implementations of methods annotated with these permissions *twice*. These two checks correspond to the two situations under which a given method could be invoked: the receiver, *this*, is at runtime either an instance of the class that the method is declared in or an instance of a subclass. Other situations cannot occur. Plural now checks both situations separately, both times encoding these fields-and-dispatch permissions with regular frame and/or virtual permissions.

- When the receiver is an instance of the class that the method is declared in then the method is in fact accessing the *virtual frame* of the receiver, i.e., the frame corresponding to the runtime type of the receiver. In this case, frame and virtual permissions in fact grant access to the same frame. Plural checks this case by using a single (frame) permission of the declared kind for accessing fields and dynamic dispatch.

- When the receiver is *not* an instance of the class that the method is declared in then frame and virtual permissions in fact grant access to *different* frames. Plural checks this case by using both a frame and a virtual permission of the declared kind, the former allowing field accesses and the latter allowing dynamic dispatch.

Calling a thus annotated method using dynamic dispatch will require a virtual permission of the declared kind at the call site. Just as with frame permissions, we assume that methods requiring permissions for both field access and dynamic dispatch are overridden. Therefore, a method can only be invoked in the second situation discussed above by a subclass through a *super*-call, for which the subclass will be required to provide separate *super*-frame and virtual permissions for the receiver as before. Implementing this extension removed 31 out of 77 false positives in checking parts of PMD (Section 7.3.2).

This extension allows checking methods *as if* the surrounding class was final. But we also allow subclassing. Figure 6.3 illustrates how a method requiring a full permission can be overridden *and called* in a subclass. (If the overridden method is never called or only requires "duplicable" permissions such as share then overriding should be straightforward.)

```
class C {
  @Full(use = Use.DISP_FIELDS)
  void m() { ... }
}
```

```
@ClassStates(@State(name = "alive", inv = "quiet == true => full(super)"))
class D extends C {
  private boolean quiet = true;
  void m() { if(quiet) { quiet = false; super.m(); quiet = true; } }
}
```

Figure 6.3: Overriding a method requiring a full "dispatch-and-fields" permission

## 6.5 Dealing with Java

This section details how Plural handles some of the Java features that are not covered explicitly by the underlying theory (cf. Chapter 4), in particular, constructors and arrays. We also discuss some aspects of Java that are *not* currently handled sufficiently. All of the features discussed in this section are common in practical programming languages; therefore, the discussion applies to languages other than Java as well.

### 6.5.1 Constructors

We leverage API implementation checking (Section 6.3.7) to reason about constructors by injecting an unpacked unique permission for the receiver frame into the initial lattice element, in addition to any explicitly required permissions for constructor arguments. Permissions needed for virtual method invocations from inside a constructor have to be declared as usual; these permissions are checked when subclass constructors invoke the superclass constructor. This means that constructors have to *admit to* any virtual method calls they perform, which appears desirable considering the amount of grief such calls can cause to subclass developers.

### 6.5.2 Private Methods

Private methods can by definition not be overridden. This means that the *frame* they work on is statically known to be the class they are defined in. Therefore, Plural handles calls to private methods akin to *super*-calls (cf. Chapter 4): frame permissions required by a private method must be satisfied with frame permissions available at the call site. This simplifies specifying the methods calling private methods because they typically do not need to require both a frame (for their own field accesses) and a virtual permission (for calling the private method) but just a frame permission.

### 6.5.3 Static Fields (Globals)

Sometimes we need to associate permissions with static fields. (Static fields are similar to global variables in procedural languages.) Plural currently simply allows doing so with permission annotations directly on the field. To simplify matters, however, static fields can only be associated with permissions that can be duplicated (more precisely, split into equally powerful permissions, i.e. share, immutable, and pure). This avoids problems when permissions from a static field access are still in use when the same static field is accessed a second time.

In order to allow unique and full permissions for static fields, one could treat static fields as fields of the surrounding "class" object, which would require passing around permissions for the surrounding

"class" objects to where their static fields are accessed. Another option would be a simple effect system that indicates which static fields are accessed in the dynamic scope of a method.

### 6.5.4 Arrays

Plural associates arrays with a permission of their own and makes sure that a modifying permission (unique, full, or share) is available upon stores into the array. Plural does *not* currently allow tracking permissions for the array elements, which is subject of future work. This issue is related to putting permissions into containers such as lists and sets.

### 6.5.5 Future Work

This section discusses how additional common language features could be treated. Plural, mostly thanks to Crystal, currently will not crash when encountering these features, but it also will not do anything special for handling them.

**Inner Classes.** Non-static inner classes in Java implicitly carry a (final) reference to an object of the surrounding class. (Local and anonymous inner) classes defined inside method bodies also implicitly hold references to (final) local variables defined in the surrounding method. Plural is currently unable to associate permissions with these implicit fields, but it appears that this is mostly a syntactic problem. Furthermore, we note that permission parameters (see Section 6.4.5) should be a good fit for these fields, since they cannot be re-assigned. Finally, Eclipse offers refactorings to promote inner classes to regular classes, making the implicit fields explicit. Plural can handle the resulting regular classes with no problem.

**Exceptions.** Exceptions to a first approximation just induce exceptional control flow paths through methods. Crystal models control flow paths of "checked" exceptions and therefore correctly propagates Plural's permission information along exceptional paths. However, this assumes that the declared post-condition of methods hold even if exceptions are thrown. This may not always be the case, as was previously discovered (Naeem and Lhoták, 2008). Plural could realistically be extended to support annotations similar to dynamic state tests annotations: for example, an exception could indicate that the receiver or parameter is in a certain state. This information could then be propagated only on exceptional branches, which Crystal's branch sensitivity allows us to do.

## 6.6 Use Cases

Plural was designed to help developers cope with API protocols. This section lists specific use cases for using Plural in the different development phases.

- *Document and understand interface protocols.* Plural's annotations can be used to formally document API protocols. Developers will be able to understand these protocols without extensively studying the informal API documentation.[13]

[13] Ongoing efforts aim at visualizing API protocols for more convenient inspection by developers.

- *Support development.* Plural can be used while developing code to ensure that it conforms to protocols of interest; both when developing clients and implementations of an API. Since the tool can be run on a single compilation unit at a time, developers can frequently re-check the code they are working on.
- *Retroactively check interface protocols.* It is also possible to check protocols in existing codebases, which is what we did in our case studies (see Chapter 7).
- *Encode heap shape and access patterns.* Even when protocols are not of interest, developers can use permissions to encode heap shape and access patterns. In particular, permissions can enforce ownership of objects (with unique or full permissions) and read-only access to objects (with immutable and pure permissions.
- *Facilitate maintenance.* Plural supports software evolution because it can re-check changed code based on the persistent developer annotations to make sure that protocols are still respected after a maintenance task.

# Chapter 7

# Evaluation

This chapter discusses a number of case studies that we performed based on Plural, the prototype protocol checking tool described in the previous chapter. These case studies evaluate protocol specification and checking:

- **Specification** of several Java standard APIs using Plural's annotations (Section 7.1).
- **Checking of compliance** to the specified APIs in two open-source codebases (Sections 7.2 and 7.3).

Section 7.4 discusses lessons learned from these case studies. Earlier versions of some of the material presented in this chapter is to appear at the ECOOP 2009 conference (Bierhoff et al., 2009).

## 7.1 Specifying APIs

This section summarizes our experience specifying Java Database Connectivity (JDBC), Collections, and several other APIs that are part of the Java standard library.

### 7.1.1 Java Database Connectivity

The Java Database Connectivity (JDBC) API defines a set of interfaces that Java programs can use to access relational databases with SQL commands. Database client applications access databases primarily through `Connection`, `Statement`, and `ResultSet` objects. Clients first acquire a `Connection` which typically requires credentials such as a username and password. Then clients can create an arbitrary number of `Statements` on a given connection. Statements are used to send SQL commands through the connection. Query results are returned as `ResultSet` objects to the client. By convention, only one result set can be open for a given statement; sending another SQL command "implicitly closes" or invalidates any existing result sets for that statement. Database vendors provide database-specific implementations of these interfaces.

This section discusses the specification of the major interfaces (including subtypes) using Plural annotations. The specified interfaces are massive: they define 440 methods, each of which is associated with about 20 lines of informal documentation in the source files themselves, for a total of almost 10,000 lines including documentation (see Table 7.1).

| JDBC interface | Lines | (Increase) | Methods | On methods | State space | Total | Mult. cases |
|---|---|---|---|---|---|---|---|
| Connection | 1259 | (9.8%) | 47 | 84 | 4 | 88 | 2 |
| Statement | 936 | (9.4%) | 40 | 64 | 2 | 66 | 0 |
| PreparedStatement | 1193 | (5.5%) | 55 | 58 | 0 | 58 | 0 |
| CallableStatement | 2421 | (5.0%) | 111 | 134 | 1 | 135 | 0 |
| ResultSet | 4057 | (15.4%) | 187 | 483 | 8 | 491 | 82 |
| **Total** | 9866 | (10.4%) | 440 | 823 | 15 | 838 | 84 |

Table 7.1: Specified JDBC interfaces with total lines, size increase due to annotations, methods, annotation counts (on methods, for defining state spaces, and total), and the use of multiple method cases in each file. The files length is almost entirely due to extensive informal documentation.

```
@States({"open", "closed"})
public interface Connection {

@Capture(param = "conn")
@Perm(requires = "share(this, open)", ensures = "unique(result) in open")
Statement createStatement() throws SQLException;

@Full(ensures = "closed")
void close() throws SQLException;

@Pure
@TrueIndicates("closed")
boolean isClosed() throws SQLException; }
```

Figure 7.1: Simplified JDBC `Connection` interface specification. Creating a statement captures a connection permission.

### Connections

The `Connection` interface primarily consists of methods to create statements, to control transactional boundaries, and a `close` method to disconnect from the database (Figure 7.1). Closing a connection invalidates all statements created with it, which will lead to runtime errors when using an invalidated statement. Due to space limits, we do not discuss our specification of transaction-related features here, but they are included in Table 7.1.

Our goal was to specify JDBC in such a way that statements and result sets are invalidated when their connections are closed. Our solution is a variant on our work with iterators (Section 3.1): we capture a share connection permission each time a statement is created on it. The captured permission has the *open* state guarantee, which guarantees that the connection cannot be closed while the statement is active. Plural releases the captured connection permission from a statement that is no longer used or when the statement is closed, as explained in Section 6.4.5. When all statements are closed then a full permission for the connection can be re-established, allowing `close` to be called.

```
@Refine({
  @States({"open", "closed"}),
  @States(refined = "open", value = {"hasResultSet", "noResultSet"}, dim = "rs") })
@Param(name = "conn", type = Connection.class, releasedFrom = "open")
public interface Statement {

@Capture(param = "stmt")
@Perm(requires = "full(this, open)", ensures = "unique(result) in scrolling")
ResultSet executeQuery(String sql) throws SQLException;

@Share("open")
int executeUpdate(String sql) throws SQLException;

@Full("open")
@TrueIndicates("hasResultSet")
@FalseIndicates("noResultSet")
boolean execute(String sql) throws SQLException;

@Capture(param = "stmt")
@Perm(requires = "full(this, open) in hasResultSet", ensures = "unique(result) in scrolling")
ResultSet getResultSet() throws SQLException;

@Full("open")
@TrueIndicates("hasResultSet")
@FalseIndicates("noResultSet")
boolean getMoreResults() throws SQLException;

@Full(ensures = "closed")
@Release("conn")
void close(); }
```

Figure 7.2: JDBC `Statement` interface specification (fragment). The captured connection parameter "conn" is released when the statement is closed or garbage-collected.

### Statements

Statements are used to execute SQL commands. Statements define methods for running queries, updates, and arbitrary SQL commands (Figure 7.2).

We specify `executeQuery` similarly to how statements are created on connections. The resulting `ResultSet` object captures a full permission to the statement, which enforces the requirement that only one result set per statement exists. Conversely, `executeUpdate` borrows a share statement permission and returns the number of updated rows. Since share and full permissions cannot exist at the same time, result sets have to be closed before calling `executeUpdate`. The `Statement` documentation implies that result sets should be closed before an update command is run, and our specification makes this point precise.

The method `execute` can run any SQL command. If it returns `true` then the executed command was a query, which we indicate with the state *hasResultSet*. `getResultSet` requires this state and returns the actual query result.

In rare cases a command can have multiple results, and `getMoreResults` advances to the next result. Again, `true` indicates the presence of a result set. We use a full permission because, like `execute` methods, `getMoreResults` closes any active result sets, as stated in that method's documentation: "Moves to this `Statement` object's next result, returns `true` if it is a `ResultSet` object, and implicitly closes any current `ResultSet` object(s) obtained with the method `getResultSet`."

Besides a plain `Statement` interface for sending SQL strings to the database, JDBC defines two other flavors of statements, "prepared" and "callable" statements. The former correspond to patterns into which parameters can be inserted, such as search strings. The latter correspond to stored procedures.

Since these interfaces are subtypes of `Statement` they inherit the states defined for `Statement`. The additional methods for prepared statements are straightforward to define with these states, while callable statements need an additional state distinction for detecting NULL cell values (similar to `ResultSet`'s `wasNull`, see below).

Overall, we were surprised at how well our approach can capture the design of the `Statement` interfaces.

#### Result sets

`ResultSet` is the most complex interface we encountered. We already discussed its most commonly used features in Chapter 1. In addition, result sets allow for random access of their rows, a feature that is known as "scrolling". Scrolling caused us to add a *begin* state besides *valid* and *end*, which represents the result set's cursor pointing "before" the first row. This allows specifying the backwards-scrolling method `previous` analogously to `next`.

Furthermore, the cell values of the current row can be updated, which caused us to add orthogonal substates inside *valid* to keep track of whether there is a *pending* update (in parallel to *read* and *unread*, see Figure 6.1).

Finally, result sets have a buffer, the "insert row", for constructing a new row. The problem is that, quoting from the documentation for `moveToInsertRow`, "[o]nly the updater, getter, and `insertRow` methods may be called when the cursor is on the insert row." Thus, scrolling methods (such as `next`) are not available while on the insert row, *although the documentation for these methods does not hint at this problem.*

Our interpretation is to give result sets two modes (i.e., states), *scrolling* and *inserting*, where the former contains the states for scrolling (shaded in Figure 1.1) as substates. `moveToInsertRow` and `moveToCurrentRow` switch between these modes (Figure 6.1). In order to make the methods for updating cells applicable in both modes we use method cases (illustrated for the method `updateInt` in Figure 6.1) which account for all 82 methods with multiple cases in `ResultSet` (see Table 7.1).

Figure 7.3 shows a fragment of the `ResultSet` interface with our actual protocol annotations. Notice how the two modes affect the methods previously shown in Figure 6.1. The figure also shows selected methods for scrolling, updating (including method cases), and inserting.[1]

### 7.1.2 Java Collections Framework

The Java Collections Framework can be seen as consisting of two parts:

[1] `updateInt` defines two cases, which are both based on a borrowed full permission. One case requires that permission in the *valid* state and ensures *pending*, while the other case requires and ensures *insert*.

```
@Refine({
  @States({"open", "closed"}),
  @States(refined = "open", value = {"scrolling", "inserting"}),
  @States(refined = "scrolling", value = {"start", "valid", "end"}, dim = "row"),
  @States(refined = "valid", value = {"read", "unread"}, dim = "access"),
  @States(refined = "valid", value = {"noUpdates", "pending"}, dim = "update"),
  @States(refined = "open", value = {"forwardOnly", "scrollable"}, dim = "scroll", marker = true),
  @States(refined = "open", value = {"readOnly", "updatable"}, dim = "cursor", marker = true) })
@Param(name = "stmt", type = Statement.class, releasedFrom = "open")
public interface ResultSet {

// changes from figure 6.1

@Full(guarantee = "scrolling", requires = "noUpdates")
@TrueIndicates("noUpdates")
boolean next() throws SQLException;

@Full(guarantee = "scrolling", requires = "valid", ensures = "read")
int getInt(int columnIndex) throws SQLException;

@Pure(guarantee = "scrolling", requires = "read", ensures = "read")
boolean wasNull() throws SQLException;

// sample additional methods

@Pure("open")
@TrueIndicates("begin")
boolean isBeforeFirst() throws SQLException;

@Full(guarantee = "open", requires = "updatable")
@Cases({
  @Perm(requires = "this in valid", ensures = "this in pending"),
  @Perm(requires = "this in insert", ensures = "this in insert")
})
void updateInt(int columnIndex, int x) throws SQLException;

@Full(guarantee = "scrolling", requires = "pending", ensures = "noUpdates")
void updateRow() throws SQLException;

@Full(guarantee = "open", ensures = "inserting")
void moveToInsertRow() throws SQLException; }
```

Figure 7.3: JDBC `ResultSet` interface specification (fragment).

- *Interfaces* defining APIs for common object containers, including lists, sets, maps, and queues, as well as the `Iterator` interface for iterating over those.

- *Standard implementations* of these interfaces with common data structures such as arrays, linked lists, hashtables, and trees.

We focused on specifying the interfaces. Following the naming convention in the API, we will use the term "collection" to include lists, sets, and queues, i.e., anything that can be iterated. Maps cannot be iterated directly; instead, their key-, value-, and entry-sets can be iterated (see below). We will use the term "container" to include collections and maps.

Many researchers, including the author, have used iterators as a case study for their specification and verification approaches (Dietl and Müller, 2005; Bierhoff, 2006; Krishnaswami, 2006; Haack and Hurlin, 2008, is an incomplete list), although most of these study only some aspects of the iterator protocol and do not necessarily define iterators in the same way as the Java standard library API.

While the protocol for using iterators to retrieve elements from the iterated collection is comparably simple (Figure 3.1), the challenge that most researchers have traditionally focused on is that of *concurrent modification*: while a collection is iterated, it must not be modified. Section 3.1 discusses the iterator protocol in more detail; Figure 7.8 shows a simplified specification with Plural annotations.

Faithful to the Java standard library, "read-only" operations that access the collection directly are permitted in our approach (Bierhoff, 2006), as does Krishnaswami (2006). On the other hand, Haack and Hurlin (2008) do not seem to allow accessing an iterated collection at all.

Studying the Java Collections API reveals a number of additional challenges.

- *Views.* The API defines a variety of operations for creating "views" on a given container. In particular, the `Map` interface provides methods `keySet`, `values`, and `entrySet` that return `Sets` of keys, values, and `Map.Entry` objects. The latter represent the key-value pairs stored in a map. One way of looking at iterators is to consider them as another kind of view (one with extremely limited interface).

- *Modifiability.* Some containers cannot be modified, i.e., elements cannot be added or removed. In particular, the Java library's `emptySet`, `emptyMap`, and `emptyList` methods return empty containers to which no elements can be added. Views and iterators can also be unmodifiable in order to avoid concurrent modification: while a collection is iterated, its views should not be used to modify the collection. And if multiple iterators over the same collection exist then none of these iterators should modify the collection, while a single iterator is allowed to remove elements from the collection being iterated.

When using access permissions it makes sense to consider concurrent modification, views, and modifiability at the same time. In particular, we used the following intuition to specify Java collections (Figure 7.4 shows these ideas at work to specify the creation of iterators over collections):

- Views including iterators "capture" a permission for the underlying container. When views and iterators are dead then the captured permission can be recovered (Section 6.4.5). For instance, iterators capture a permission for the iterated over ("underlying") collection.

- If the view or iterator wants to modify the container it will capture a full permission (this could be relaxed to a share permission).

```
public interface Iterable<T> {

  @Capture(param = "underlying")
  @Cases({
    @Perm(requires = "full(this)", ensures = "unique(result) in modifiable"),
    @Perm(requires = "immutable(this)", ensures = "unique(result) in readonly")
  })
  Iterator<T> iterator();

}
```

Figure 7.4: Method cases allow iterators to be read-only or modifiable at the client's choice

- Read-only (unmodifiable) views and iterators will capture an immutable permission for the container.
- Method cases "delay" the choice between modifiable and read-only views until the client forces a decision by modifying the view or creating a second view (forcing both to be immutable).[2] For instance, creating an iterator can be defined with two cases, one capturing a full and one capturing an immutable collection permission. If a full collection permission is available when `iterator` is called then both options will be carried forward as possible choices until one of them becomes inconsistent (see Section 6.4.3).
- Modifiability for containers and views is distinguished using two *marker states* (see Section 6.4.4), "modifiable" and "readonly", as used in the two method cases in Figure 7.4.

These rules reveal an interesting interplay between states and permissions for encoding modifiability. A modifying permission such as full does not guarantee that the referenced view or iterator can be used to modify a container: the permission also needs the "modifiable" typestate. This typestate will imply that the view or iterator has a full permission for the underlying container (with "modifiable" typestate), which is needed to modify the underlying collection.

### Deep Collections

Our work with PMD (Section 7.3) motivated a closer look at collection *nesting*. PMD frequently uses containers as elements in other containers, such as lists of lists or maps of sets. In these cases, we need to track permissions for the elements of the outer container in order to access the inner one.

Similar to Haack and Hurlin (2008), we can do so by distinguishing "shallow" and "deep" containers (again using marker states). Deep containers hold permissions for their elements, while shallow ones *do not* hold on to permissions for their elements. In general, one could want to store any kind of permission for the elements of a container, but in PMD we were mostly interested in keeping unique permissions for "shallow" containers in a "deep" container.

This idea has important consequences: whenever elements are added to a deep container, the appropriate permission for the new element must be available. (Figure 7.5 illustrates this for a list that keeps unique permissions when it is deep.) More problematic, however, is how to access the elements in a container.

[2] We originally presented this idea in Bierhoff (2006).

```
@States(value = {"shallow", "deep"}, marker = true)
public interface List<T> {
  @Full
  @Cases({
    @Perm(requires = "this in shallow"),
    @Perm(requires = "this in deep * unique(#0)")
  })
  public boolean add(T e);

  @Lend
  @Cases({
    @Perm(requires = "pure(this) in shallow"),
    @Perm(requires = "unique(this) in deep", ensures = "unique(result)"),
    @Perm(requires = "immutable(this) in deep", ensures = "immutable(result)")
  })
  public T get(int index);
}
```

Figure 7.5: List interface that optionally maintains permissions for contained elements. "#0" in `@Perm` annotations refers to the first argument of the annotated method

A method such as `get` (for retrieving an element from a list or map) should return, for deep containers, a permission for the contained object so clients can call methods on that object.

But now the container invariant is in violated: while the client is working with the element returned from `get`, the container does not have the permission for this element.[3] What has to happen, conceptually, is that the element's permission is returned to the container before the container is used again. We can enforce this by borrowing the element permission "from" the container (Boyland et al., 2007), temporarily consuming the container permission until the element is no longer used (with the `@Lend` annotation in Figure 7.5). This is similar to how we capture a container permission in its view and release it when the view is no longer used, and in fact the implementation in Plural is identical. As detailed in Boyland et al. (2007), this is legal because the *lender*, the container, remains unpacked until the lending is over, which allows it to violate its invariant[4].

But what if we borrow an element from a deep container for which we have an immutable permission? Surely we cannot expect a unique permission for the borrowed element, since other permissions might be used to borrow the very same element. In this case, we will only get a immutable permission for the borrowed element as well (Figure 7.5). This, in particular, also applies to "readonly" views and iterators.

We heavily use method cases for specifying container methods. They can express the different behavior of many container methods depending on whether a unique or immutable permission is available (e.g., they create views in different states and return different permissions for contained elements).

**Outlook.** While we limited our discussion here to two kinds of collections, shallow and deep, there is no fundamental reason why we could not use more method cases (and corresponding marker states) to put

[3] This appears to only be an issue with unique and full permissions; the container could split immutable, share, and pure permissions into two, retain one of them, and give the other one to the client.

[4] Unpacking is equivalent to what is called "carving" in Boyland et al. (2007).

other permissions for each element into a container. In fact, we did so for PMD. There, we distinguish "quietElems" collections besides shallow and deep ones, which contain full permissions in the "quiet" state. This is, of course, far less convenient than being able to "parameterize" a container with the permission it keeps for each of its elements, and our experience here suggests that such a parametrization is in fact straightforward, although not supported by Plural at this time.

Although not pursued in this dissertation, we believe that the idea of "deep" collections can be adapted for tracking permissions for array elements.

### 7.1.3 Other Java Standard Libraries

**Regular expressions.** The API includes two classes. A `Pattern` is created based on a given regular expression string. Then, clients call `find` or `match` to match the pattern in a given string, both of which return an instance of `Matcher`. The `Matcher` instance returned by these operations can be used to retrieve details about the current match and to find the next matching substring.

We easily specified this protocol in Plural. As with iterators, we capture a immutable `Pattern` permission in each `Matcher`. We use a typestate *matched* to express a successful match and require it in methods that provide details about the last match.

**Streams.** Streams are discussed in detail in Section 3.2. Their protocol is straightforward to specify with Plural's annotations. We point out that "buffered" streams and other kinds of wrappers end up "capturing" a permission for the wrapped stream. There should only be one wrapper per stream (which we can enforce by capturing a full permission in them), but wrappers are nested inside one another in practice, which is no problem for our approach.

**Exceptions.** When creating an exception, a "cause" (another exception) can be set *once*, either using an appropriate constructor or, to our surprise, using the method `initCause`. The latter retrofits causes into legacy exceptions defined before exception causes were introduced in Java 1.4 (Figure 7.6). This protocol is trivial to specify in Plural, but it was fascinating that even something as simple as an exception has a protocol.

## 7.2 Beehive: Verifying an Intermediary Library

This section summarizes a case study in using Plural for checking API compliance in a third-party open source codebase, Apache Beehive.

Beehive[5] is an open-source library for declarative resource access. We have focused on the part of Beehive that accesses relational databases using JDBC. Beehive clients define Java interfaces and use Java annotations to associate the SQL command to be run when methods in these interfaces are called. Notice that this design is highly generic: the client-specified SQL commands can include parameters that are filled with the parameters passed to the associated method. Beehive then generates code stubs implementing the client-defined interfaces that simply call a generically written SQL execution engine, `JdbcControl`, whose implementation we discuss below.

[5] `http://beehive.apache.org/`

```
/**
 * Initializes the <i>cause</i> of this throwable to the specified value.
 * (The cause is the throwable that caused this throwable to get thrown.)
 *
 * <p>This method can be called at most once. It is generally called from
 * within the constructor, or immediately after creating the
 * throwable. If this throwable was created
 * with {@link #Throwable(Throwable)} or
 * {@link #Throwable(String,Throwable)}, this method cannot be called
 * even once.
 *
 * @param cause the cause (which is saved for later retrieval by the
 * {@link #getCause()} method). (A <tt>null</tt> value is
 * permitted, and indicates that the cause is nonexistent or
 * unknown.)
 * @return a reference to this <code>Throwable</code> instance.
 * @throws IllegalArgumentException if <code>cause</code> is this
 * throwable. (A throwable cannot be its own cause.)
 * @throws IllegalStateException if this throwable was
 * created with {@link #Throwable(Throwable)} or
 * {@link #Throwable(String,Throwable)}, or this method has already
 * been called on this throwable.
 * @since 1.4
 */
public synchronized Throwable initCause(Throwable cause) {
    if (this.cause != this)
        throw new IllegalStateException("Can't overwrite cause");
    if (cause == this)
        throw new IllegalArgumentException("Self-causation not permitted");
    this.cause = cause;
    return this;
}
```

Figure 7.6: Java exceptions have an initialization protocol! Verbatim copy from the Java 6 standard library source code for `java.lang.Throwable` included with JDK 1.6.0_04.

We first describe the APIs used by Beehive before discussing the challenges in checking that Beehive correctly implements a standard Java API and its own API.

### 7.2.1 Checked Java Standard Library APIs

We checked protocols of four Java standard APIs used by Beehive, highlighting Plural's ability to treat APIs orthogonally.

1. **JDBC.** We described the JDBC specification in Section 7.1.1. Since Beehive has no apriori knowledge of the SQL commands being executed (they are provided by a client), it uses the facilities for running "any" SQL command described in Section 7.1.1. Its use of result sets is limited to reading cell values, and a new statement is created for every command. We speculate that the Beehive developers chose this strategy in order to ensure that result sets are never rendered invalid from executing another SQL command, which ends up helping our analysis confirm just that.

2. **Collections API.** Beehive generically represents a query result row as a map from column names to values. One such map is created for each row in a result set and added to a list which is finally returned to the client. Section 7.1.2 discusses our specification of maps and lists in detail.

3. **Regular expressions.** Regular expressions (see Section 7.1.3) are only used once in Beehive. The pattern being matched is a static field in one of Beehive's classes, which we annotate with `@Imm`.

4. **Exceptions.** Beehive uses `initCause` (see Figure 7.6) to initialize a legacy exception that is part of the Java standard library, `NoSuchElementException` (which, incidentally, is the exception prescribed by the Java standard library to signal a iterator protocol violation). It is unknown why the library designers did not retrofit the "cause" mechanism into their own exceptions, but Beehive, due to this omission, has no choice but to use `initCause`.

### 7.2.2 Implementing an Iterator

Beehive implements an `Iterator` over the rows of a result set. Figure 7.7 shows most of the relevant code. We use state invariants, i.e., predicates over the underlying result set (see Section 6.3.7), to specify iterator states. Notice that alive is our default state that all objects are always in. Thus its state invariant is a conventional class invariant (Leavens et al., 1999; Barnett et al., 2004) that is established in the constructor and preserved afterwards.

When checking the code as shown, Plural issues 3 warnings in `hasNext` (see Table 7.2). This is because our vanilla iterator specification (Bierhoff and Aldrich, 2007b) assumes `hasNext`, which tests if an element can be retrieved, to be pure. Beehive's `hasNext` is not pure because it calls `next` on the `ResultSet` (Figure 7.7).

This problem can be fixed, for example, by advancing the result set to the next row at the end of the iterator's `next` method (after constructing the return value) and remembering the outcome in the existing flag. The iterator constructor can initialize the flag by moving the result set to the first row, if it exists. That way, `hasNext` is pure because it only tests whether the flag is `true`. However, this code change has the disadvantage that the result set may be unnecessarily advanced to a row that is never retrieved with a

subsequent call to the iterator's `next` method. Furthermore, it duplicates the code for advancing to the next row in `next` and the constructor.

Alternatively, the warnings disappear when we change `hasNext`'s specification to use a full permission.

Also, note that `next`'s specification requires *available*, which guarantees that `_primed` is `true` (see Figure 7.7). This makes the initial check in `next` superfluous (if all iterator clients were checked with Plural as well).

### 7.2.3 Formalizing Beehive Client Obligations

Beehive is an intermediary library for handling resource access in applications: it uses various APIs to access these resources and defines its own API through which applications can take advantage of Beehive. We believe that this is a very common situation in modern software engineering: application code is arranged in layers, and Beehive represents one such layer. The resource APIs, such as JDBC, reside in the layer below, while the application-specific code resides in the layer above.

Beehive's API is defined in the `JdbcControl` interface, which is implemented in `JdbcControlImpl`. `JdbcControlImpl` in turn is a client to the JDBC API. The class provides provides three methods `onAcquire`, `invoke`, and `onRelease` to clients. The first one creates a database connection, which the third one closes. `invoke` executes an SQL command and, in the case of a query, maps the result set into one of several possible representations. One representation is the iterator mentioned above; another one is a conventional `List`. Each row in the result is individually mapped into a map of key-value pairs (one entry for each cell in the row) or a Java object whose fields are populated with values from cells with matching names.

Notice that some of these representations, notably the iterator representation, of a result require the underlying result set to remain open. The challenge now is to ensure that `onRelease` is not called while these are still in use because closing the connection would invalidate the results. This requirement is identical to the one we described for immediate clients of `Connection`, and thus we should be able to specify it in the same way.

However, the connection is in this case a field of a surrounding Beehive `JdbcControlImpl` object, and Plural has currently no facility for letting `JdbcControlImpl` clients keep track of the permission for one of its fields. Therefore, we currently work with a simplified `JdbcControlImpl` that always closes result sets at the end of `invoke`. Its specification, as desired, enforces that `onAcquire` is called before `onRelease` and `invoke` is only called "in between" the other two. This, however, means that our simplified `JdbcControlImpl` does not support returning iterators over result sets to clients, since they would keep result sets open. Overcoming this problem is discussed in the next section.

As mentioned, Beehive generates code that calls `invoke`. The generated code would presumably have to impose usage rules similar to the ones for `invoke` on *its* clients. Plural could then be used to verify that the generated code follows `JdbcControlImpl`'s protocol.

### 7.2.4 Overhead: Annotations in Beehive

The overhead for specifying Beehive is summarized in Table 7.2. We used about 1 annotation per method and 5 per Beehive class, for a total of 66 annotations in more than 2,000 lines, or about one annotation every 30 lines. Running Plural on the 12 specified Beehive source files takes about 10 seconds.

```
@ClassStates({
  @State(name="alive",
    inv="full(_rs,scrolling) && full(_rowMapper) in init && _primed == true => _rs in valid"),
  @State(name="available", inv="_primed == true") })
@NonReentrant
public class ResultSetIterator implements java.util.Iterator {
private final ResultSet _rs;
private final RowMapper _rowMapper;
private boolean _primed = false;

/** @return true if there is another element */
@Pure(guarantee = "next", fieldAccess = true)
@TrueIndicates("available")
public boolean hasNext() {
    if (_primed) {
        return true;
    }

    try {
        _primed = _rs.next();
    } catch (SQLException sqle) {
        return false;
    }
    return _primed;
}

/** @return The next element in the iteration. */
@Full(requires = "available", ensures = "hasCurrent", fieldAccess = true)
public Object next() {
    try {
        if (!_primed) {
            _primed = _rs.next();
            if (!_primed) {
                throw new NoSuchElementException();
            }
        }
        // reset upon consumption
        _primed = false;
        return _rowMapper.mapRowToReturnType(/* analysis-only */ _rs);
    } catch (SQLException e) {
        // Since Iterator interface is locked, all we can do
        // is put the real exception inside an expected one.
        NoSuchElementException xNoSuch = new NoSuchElementException("ResultSet exception: " + e);
        xNoSuch.initCause(e);
        throw xNoSuch;
    }
} }
```

Figure 7.7: Beehive's iterator over the rows of a result set (constructor omitted). Plural issues warnings because `hasNext` is impure.

| Beehive class | Lines | Methods | Annotations Meths. | Invs. | Total | Plural warnings | False pos. |
|---|---|---|---|---|---|---|---|
| DefaultIteratorResultSetMapper | 37 | 2 | 1 | 0 | 1 | 0 | 0 |
| DefaultObjectResultSetMapper | 127 | 2 | 2 | 0 | 2 | 0 | 0 |
| JdbcControlImpl | 521 | 13 | 13 | 1 | 14 | 2 | 1 |
| ResultSetHashMap | 85 | 9 | 9 | 0 | 9 | 0 | 0 |
| ResultSetIterator | 106 | 4 | 4 | 3 | 7 | 3 | 0 |
| ResultSetMapper | 32 | 2 | 2 | 0 | 2 | 0 | 0 |
| RowMapper | 260 | 5 | 9 | 1 | 10 | 0 | 0 |
| RowMapperFactory | 156 | 7 | 3 | 0 | 3 | 4 | 4 |
| RowToHashMapMapper | 57 | 2 | 4 | 1 | 5 | 0 | 0 |
| RowToMapMapper | 49 | 2 | 4 | 1 | 5 | 0 | 0 |
| RowToObjectMapper | 236 | 3 | 4 | 0 | 4 | 0 | 0 |
| SqlStatement | 511 | 14 | 4 | 0 | 4 | 0 | 0 |
| **Total** | 2158 | 65 | 59 | 7 | 66 | 9 | 5 |

Table 7.2: Beehive classes checked with Plural. The middle part of the table shows annotations (on methods, invariants, and total) added to the code. The last 2 columns indicate Plural warnings and false positives.

### 7.2.5 Analysis Precision

Plural reports 9 problems in Beehive. Three of them are due to the impure `hasNext` method in `ResultSetIterator` (see Section 7.2.2). Letting `hasNext` use a full permission removes these warnings. Another warning in `JdbcControlImpl` is caused by an assertion on a field that arguably happens in the wrong method: `invoke` asserts that the database connection is open before delegating the actual query execution to another, "protected" method that uses the connection. Plural issues a warning because a subclass could override one, but not the other, of these two methods, and then the state invariants may no longer be consistent. The warning disappears when moving the assertion into the protected method. Furthermore we note that our state invariants guarantee that the offending runtime assertion succeeds.

The remaining warnings issued by Plural are false positives. This means that our false positive rate is is around 1 per 400 lines of code. We consider this to be quite impressive for a behavioral verification tool applied to complicated APIs (JDBC and others) and a very challenging case study subject (Beehive).

#### Sources of Imprecision

The remaining warnings in Beehive fall into the following categories:

- *Reflection (1).* Plural currently does not give permissions to objects created using reflection, which Beehive uses in `RowMapperFactory`.

- *Static fields (3).* `RowMapperFactory` manipulates a static map object, which we specified to require full permissions. For soundness, we only allow duplicable permissions, i.e., share, pure, and immutable, on static fields.

- *Complex invariant (1).* `JdbcControlImpl` opens a new database connection in `onAcquire` only if one does not already exist. Plural currently cannot capture the invariant that a non-`null` field implies a permission for that field, which would allow Plural to verify the code.

These are common sources of imprecision in static analyses. We are considering tracking fields as implicit parameters in method calls, as discussed in Section 7.2.3, and static fields could be handled in this way as well. Related to this issue is also a place in Beehive where a result set that was assigned to a field in the constructor is implicitly passed in a subsequent method call. We turned it into an explicit method parameter for now (the call to `mapRowToReturnType` in Figure 7.7). Java(X) has demonstrated that fields can be tracked individually (Degen et al., 2007), although we would like to track permissions for "abstract" fields that do not necessarily correspond to actual fields in the implementation. We are also working on a strategy for handling object construction through reflection, and on generalizing the state invariants expressible in Plural.

We also simplified the Beehive code in a few places where our approach for tracking local aliases leads to analysis imprecisions. Since local alias tracking is orthogonal to tracking permissions we used the simplest available, sound solution in Plural, which is insufficient in some cases. We plan to evaluate other options.

Problems occur when the same variable is assigned different values on different code paths, usually depending on a condition. When these code paths rejoin, Plural assumes that the variable could point to one of several locations, which forbids strong updates. We are investigating using more sophisticated approaches that avoid this problem. Alternatively, Plural will work fine when the part of the code that initializes a variable on different paths is refactored into a separate method. Notice, however, that tracking local aliasing is a lot more tractable than tracking aliasing globally. Permissions reduce the problem of tracking aliasing globally to a local problem.

Furthermore, we modified Beehive in one place to not use correlated `if`s (in a loop), which the tool currently does not support. This required moving a dynamic state test and inserting a "break" statement into a loop.

Finally, we assumed one class to be non-reentrant, but we believe a more complicated specification would allow the class to be analyzed assuming reentrancy (using intermediate states as seen in Section 3). Therefore, we use the (currently unchecked) annotation shown in Figure 7.7 to mark a class as non-reentrant, which causes Plural to omit certain checks during API implementation checking. We are planning on checking this annotation with Plural in the future.

#### Refactoring option

Notice that besides improving the tool there is usually the option of refactoring the problematic code. We believe that this is an indicator for the viability of our approach in practice, independent of the features supported by our tool: developers can often circumvent tool shortcomings with (fairly local) code changes. On the other hand, we have not seen many examples that fundamentally could not be handled by our approach.

## 7.3 Iterators in PMD: Scalability and Precision

We used version 3.7 of PMD as it is included in the DaCapo 2006-10-MR2 benchmarks[6] to investigate how Plural can be used to check existing large codebases. We chose this codebase because it was used as a benchmark for recent whole-program protocol analyses based on *tracematches* (Bodden et al., 2008; Naeem and Lhoták, 2008), which were among others used to check two protocols related to iterators in PMD. This case study investigates the use of Plural for checking similar protocols in the same codebase.

Unlike previous modular program verification tools, whole-program analyses promise aliasing flexibility, and tracematch-based analyses are the only ones we are aware of that can automatically check protocols involving object dependencies (such as the "concurrent modification" protocol discussed below). Therefore, this case study allows direct comparison between ours and the leading approaches with comparable expressiveness.

In particular, we focused on two well-known protocol errors related to iterators over collections (see Section 3.1):

- *Iterator usage.* It is only legal to call `next` on an iterator that has another element available. Usually, this requires calling *and checking the return value of* the iterator's `hasNext` method, although other kinds of dynamic state tests are possible.

  Tracematch-based analyses report high precision in checking this protocol, and our case study attempts to establish how much overhead our approach imposes when checking such relatively simple protocols (Section 7.3.1).

- *Concurrent modification.* Collections must not be modified directly while they are iterated. If collections are modified through an iterator (by, for instance, removing an element) then the collection must not be iterated over with other iterators. These represent changes of a collection without the knowledge of its iterators, which are forbidden because changes to a collection may invalidate iterators' "pointers" into the collection. These errors are called "concurrent modification" even though they can easily occur in single-threaded programs.

  Tracematch-based analyses report low precision in checking this protocol in PMD, and so we were interested in seeing whether our approach could provide added benefit (Section 7.3.2).

After discussing our own experiences checking these protocols in PMD, Section 7.3.3 compares our result to previous Tracematch-based analyses.

Case studies with large codebases are fundamentally challenging to do with a tool like Plural because providing annotations for methods and classes throughout the codebase may require considerable manual effort. This complicates case studies in practice. But it also enables more realistic and detailed estimates of the overhead on developers imposed by modular approaches.

**Background on PMD.** PMD is a bug-finding tool primarily for Java code. It comes with a plethora of "rules" that can be used to expose common problems in Java source files. Some of these rules look for bugs, but others are targeted at enforcing coding conventions such as avoiding large methods or unused local

[6] `http://dacapobench.org/`

variables. The version of PMD used in this case study consists of close to 40 KLOC (excluding comments and blank lines) in almost 450 classes.

PMD supports several modes of use including as an application that can be started from the command line to check a given set of rules in a given set of source files, as a "task" in an automated build script, to run "benchmarks", and as part of a GUI application. A typical "run" consists of the following steps:

- Collect source files and rules to be checked against each other.
- For each source file:
  - Parse the file and create a structured representation of it as an Abstract Syntax Tree (AST).
  - Check each rule, using the AST. (Most rules are simple tree walkers that visit the AST.)
- "Render" rule violations in the desired format, e.g., as an HTML page.

PMD makes heavy use of collections and iterations. Many collections are declared as fields in another class, while iterations typically happen locally to a method. Many collections are long-lived in that they are created close to the start of the program and continue to be used until its end.

### 7.3.1 Iterator Usage Protocol

Iterators are widely used in PMD, and most iterations in PMD are over Java Collections (see Section 7.2.1), but PMD implements a few iterator classes over its own data structures as well. There are 170 distinct call sites invoking the `next` method of the iterator interface in the codebase.

It was very easy to check with Plural that the protocol for using iterators was followed in PMD. We simply used the simplified iterator specification shown in Figure 7.8 to ensure that PMD

- calls and tests `hasNext` before every call to `next`, and
- calls `remove` at most once after each call to `next`.

#### Overhead and Precision

Plural can verify that that 167 out of 170 calls to `next` call and test `hasNext` first. `remove` is only called in one place, which Plural verifies as well.

It took the author about 75 minutes to check that this protocol is followed in all of PMD, using only 15 annotations. Most iterator usages could be verified by Plural without any user intervention because they are created and used inside one method. Annotations were needed where iterators were returned from a method call inside PMD and then used elsewhere. In one place an iterator is passed to a helper method *after* checking `hasNext`, and we could express the contract of this helper method with a suitable annotation.[7]

Three calls to `next` could not be verified in this case study, although they represent correct usage: in these cases, PMD checks that a set is non-empty before creating an iterator over the tested set and immediately retrieving (only) the first element. This pattern works around a shortcoming in Java's Set interface—the

[7] This method trips up one of the previous analyses (Bodden et al., 2008).

```
@Refine({
  @States(value = {"available", "end"}, dim = "next"),
  @States(value = {"retrieved", "removed"}, dim = "current")
})
public interface Iterator<T> {

  @Pure("next")
  @TrueIndicates("available")
  public boolean hasNext();

  @Full(requires = "available", ensures = "retrieved")
  public T next();

  @Full(guarantee = "current", requires = "retrieved", ensures = "removed")
  public void remove();

}
```

Figure 7.8: Simplified annotations for checking iterator protocol

absence of a method to get "some" element from the set. (The only way to get to the elements of a set is to iterate over them.) We could not verify these calls to `next` because the collection being iterated is not considered in our simplified iterator protocol. However, when the collection is also tracked with Plural (see below) then we can verify this pattern.

In summary, checking the iterator protocol in PMD using Plural was exceedingly easy and imposed almost no overhead. Running Plural on PMD's entire source tree of 446 files (with the same configuration as for Beehive) takes about 4 minutes.

### Iterator implementations

PMD implements three iterators of its own. In one of them, `TreeIterator`, the implementation of `hasNext` is not only impure, like Beehive's iterator, but advances the iterator every time it is called. Thus, failure to call `next` after `hasNext` results in lost elements. The other iterators exhibit behavior compatible with the conceptual purity of `hasNext`: `next` is used to pre-fetch the element to be returned the *next* time it is called before returning the current element. `hasNext` then simply checks the pre-fetched element is valid, which is typically a pure operation.

In light of these and the iterator implementation in Beehive (Figure 7.7), it appears legitimate to ask whether `hasNext` is really a pure operation. This would have significant consequences for behavioral specification approaches like the JML (Leavens et al., 1999) or Spec# (Barnett et al., 2004) because they use pure methods in specifications. Conventionally, the specification of `next` in the JML would be "`requires hasNext()`", but that would be illegal if `hasNext` was not pure. In contrast, our specifications are more robust to the non-purity of `hasNext`. In fact, Plural can verify iterator usage in PMD with a full permission for `hasNext` with the same precision.

### 7.3.2 Concurrent Modifications

After the initial success of checking the "shallow" iterator protocol (see previous section) we decided to extend the PMD case study to check for "concurrent modifications". This turned out to be challenging for several reasons:

- Collections are often held in fields, and some of them are long-lived. Collections are also often used as method parameters or return values. In all these these situations Plural requires annotations in order to check most collection accesses. Overall, PMD declares 447 lists and sets and 139 maps as fields, local variables, method parameters, or method return types.

- Additionally, collections are often nested: For example, we found a map from keys to maps from other keys to integers. Lists of lists, maps from keys to lists, etc., were also very common. These pose significant challenges that are related to tracking permissions for array cells (see Section 7.1.2).

In order to cope with the many collections to be tracked we limited the case study to the following core parts of PMD:

- The handling of rules and source files to be analyzed, including maps of properties for configuring individual rules.

- The rules for checking Java source files (thereby excluding rules for other types of files).

- The handling and reporting of rule violations.

These represent many of the collections being used in a typical PMD run to check Java source files against some set of rules, with one important exception: we did not attempt to check PMD's parsing and binding infrastructure, or the related AST implementation. This puts a caveat on our checking of "rules", as will be discussed below.

#### Overhead

Table 7.3 summarizes the number of annotations used for checking concurrent modification problems in PMD, broken down by top-level packages. Notice that the largest package, `pmd.ast`, is almost entirely automatically generated by a parser generator. Almost all of the annotations in this package are on the more than 100 methods defined in the AST *visitor* interface (2 annotations per method), which we needed to annotate in order to be able to track the many collections stored in fields of the various *rules* (most of which are AST visitors).

Focusing on the packages that we considered in detail as part of this case study, we used on average about 1 annotations for every two methods, or 1 annotation per 25 lines of code. It took the author about 18 hours to provide these annotations, which includes the time to annotate Java Collections classes as needed. The code was refactored in 12 places to allow verification with Plural, mostly to get around limitations of Plural's aliasing analysis and inheritance handling (see Section 7.2.5).

| Package | Considered impl. | KLOC | Annotations | Remaining warnings Initially | Improved |
|---|---|---|---|---|---|
| pmd | Yes | 2.2 | 160 | 13 | 13 |
| pmd.renderers | Yes | 0.7 | 13 | 0 | 0 |
| pmd.rules.** | Yes | 5.4 | 156 | 58 | 33 |
| pmd.sourcetypehandlers | Yes | 0.1 | 4 | 0 | 0 |
| pmd.stat | Yes | 0.2 | 12 | 7 | 0 |
| **Considered** | Yes | 8.7 | 345 | 78 | 46 |
| pmd.ant | No | 0.3 | 0 | 9 | 9 |
| pmd.ast | No | 14.8 | 235 | 3 | 3 |
| pmd.cpd.** | No | 4.2 | 7 | 41 | 41 |
| pmd.dfa.** | No | 1.7 | 8 | 65 | 65 |
| pmd.jaxen | No | 0.4 | 7 | 6 | 6 |
| pmd.jsp.** | No | 6.4 | 0 | 28 | 28 |
| pmd.parsers | No | 0.05 | 0 | 0 | 0 |
| pmd.quickfix | No | 0.01 | 0 | 0 | 0 |
| pmd.symboltable | No | 1.1 | 8 | 52 | 52 |
| pmd.util.** | No | 1.7 | 7 | 21 | 21 |
| **Not considered** | No | 30.8 | 272 | 225 | 225 |
| **Total** | Both | 39.4 | 617 | 303 | 271 |

Table 7.3: PMD annotations for preventing concurrent modifications. Package names with ** include sub-packages. The improvements to the number of remaining warnings come from support for permissions allowing dispatch and field access (Section 6.4.7)

### Sources of Imprecision

We are not aware of any iterator-related bugs in PMD, and so all the remaining warnings that Plural issues have to be considered false positives. While the presence of false positives in the parts of PMD whose implementation we did not consider as part of the case study (see Table 7.3) are not surprising, this section discusses in detail the origin of the remaining warnings in the parts of PMD whose iterator and collection usage we attempted to verify.

When we first used Plural for checking PMD we did not have support for permissions allowing both dynamic dispatch and field access as described in Section 6.4.7. As a result, 42 out of 78 warnings (second to last column in Table 7.3) were due to methods accessing fields and making dynamically dispatched method calls. This and several other experiences motivated adding support for "dispatch-and-fields" permissions to Plural (Section 6.4.7). As a result, Plural now only issues 46 warnings in the parts of PMD we considered.

- *Captured permission tracking (13/46).* Plural loses precision about permissions other than unique that are captured in loops, which could be avoided with smarter lattice comparison and join implementations.

- *Foreign field access (7/46).* One rule in PMD accesses fields of objects other than the receiver, which Plural currently does not support. We believe that these direct field accesses could be refactored into method calls.

- *Unpacking (5/46).* Unpacking an immutable permission in Plural currently yields pure permissions for fields, but this leads to imprecisions when immutable permissions for fields are needed. Plural should yield immutable permissions (as in Boyland et al., 2007) for fields associated with unique or full permissions, which would remove these warnings.

- *Unreachable code (4/46).* One Java rule violates method pre-conditions in case another method returns a non-null value, but it appears that **null** is always returned. We believe that these unreachable code blocks could be detected and ignored by Plural.

- *Static fields (3/46).* Static fields lead to problems similar to our experience with Beehive (see Section 7.2.5).

- *Superclass fields (2/46).* Plural currently does not allow directly accessing superclass fields in subclasses. Unpacking the object to the accessed frame should remove this problem.

- *Array, reflection, library call (3/46).* Array access, use of reflection, and a call into a XML library we did not specify cause one warning each.

- *Miscellaneous (9/46).* The remaining errors have various causes including a place where elements of a "deep" list are moved into another list one by one, which our specification of collections does not support.

This discussion shows that the vast majority of false positives come from insufficient support for inheritance as well as a variety of tool shortcomings. We believe that a more mature tool could avoid most of the problems we found.

### Handling the rest of PMD

As Table 7.3 suggests, we excluded the automatically generated Java AST (package pmd.ast), PMD's rules related to JSPs (pmd.jsp) and C++ code (pmd.cpd), its dataflow analysis framework (pmd.dfa), and its graphical user interface (pmd.util) from consideration, as well as the already mentioned binding infrastructure (pmd.symboltable), which in particular includes "scope" classes for maintaining binding information. Except for scopes, the excluded parts of PMD are independent from the parts we considered in this case study. Moreover, these excluded parts seem to have many similarities with the parts we considered since they essentially contain "rules" for other programming languages.

The Java rules in PMD contain 9 iterations over keys or entries of variable declaration maps and 3 iterations over keys or entries of method declaration maps. These maps come from "scope" objects that represent essentially type bindings in PMD's AST. We annotated the respective methods to return permissions for the maps, but since we excluded the AST and binding information, i.e., the scopes, from our case study, it is not guaranteed by Plural that the maps are not modified while rules iterate through their entries or keys.

As we will see in the next section, maps are not considered in the tracematch-based analyses that we aim to compare Plural with (although 4 of the iterations in question nonetheless apparently confuse Bodden's analysis). Therefore, excluding scopes has limited impact on our comparison. Nonetheless, we discuss applying Plural to them below, since they reveal an interesting tooling limitation.

We manually inspected the code and determined that these maps are not modified when rules are executing. This is because PMD follows the following steps in processing a given Java source file.

1. *Parsing.* Parsing creates AST objects and fixes their references to their respective parents and children. The parser code is generated using the parser generator jjTree.

2. *Find declarations.* A AST visitor is used to create the tree of scopes and find all declarations in a given AST. AST node fields are set to point to their scopes, and scopes reference their parents. Declarations are added to the nearest enclosing scope when they are encountered by the visitor. After this pass over the AST, the scope tree is fixed.

3. *Bind name occurrences to declarations.* A second AST visitor binds names in the AST to their declarations and modifies the declarations to reference all their occurrences. Only after this pass, nodes and scopes are fully initialized and not modified subsequently.

4. *Rule checking.* Now the rules are run, which do not modify AST nodes or scopes but use information from scopes about name occurrences of given declarations as well as the methods declared in a class, as mentioned above. Rules are also typically implemented as AST visitors.

Thus PMD clearly follows a protocol in building up and manipulating AST nodes and scopes in several phases, before running rules over the now immutable AST. Notice that this intuition is not currently enforced in the PMD codebase: nothing prevents rules from changing or deleting the binding information since scopes are defined with one interface `Scope` that allows both manipulating binding information and retrieving the resulting maps.

Scopes form a tree with parent pointers (but without child pointers), and AST nodes likewise form a tree with parent pointers. Some, but not all, AST nodes have scopes associated with them, and the two tree structures are parallel to each other.

Tree structures are a well-known challenge for program verification methodologies that currently require manual (but sometimes automatically checkable) proofs.[8] We did develop a way of handling tree structures, such as the tree of scopes built up by PMD (Bierhoff and Aldrich, 2008), but Plural is not equipped to apply this solution automatically. The idea is to distribute permissions for tree nodes amongst its parent and children and assemble them when nodes need to be modified. Among other things, this requires an invariant that the permissions in a list of children combined together represent a full permission for a given node, which we cannot express with Plural's annotations. This also requires unpacking multiple objects at once, which Plural does not permit (see Section 6.3.7).

We have at least two other choices for enforcing that scopes are only manipulated before rules start iterating the bindings.

- We could define a typestate-based protocol for scopes. We would have to use share permissions for referencing scopes (since they are aliased through all their children), which would make Plural require dynamic state tests when calling at least some of the scope methods.
- A third option is to leverage related research results. In particular, Dean Sutherland has successfully used thread colors (Sutherland, 2008) to enforce execution phases. In his work, phases are associated with different thread colors, and the scope methods would require the respective color. (Notice the similarity to the previous solution.) It appears that thread coloring would not be able to enforce that binding maps are indeed immutable when iterated. This leaves the author wondering whether the two approaches could be beneficially (and soundly) combined to leverage their strengths and compensate their weaknesses.
- Finally, one could think about think about special support for traversing trees. For instance, when visiting a node, one could (implicitly or explicitly) have permissions for all parents available. (This is essentially the solution that separation logic offers (Jacobs and Piessens, 2008).) This also leads to the observation that tree traversals only seem to occur in certain kinds of programs, for instance, compilers, for which more specialized solutions may be appropriate.

In summary, the PMD scopes are hard to handle and may in fact constitute a interesting challenge problem for future research in this area.

### 7.3.3 Comparison to Tracematches

The two protocols we studied were also checked with state-of-the-art whole-program protocol checkers based on Tracematches in the same version of PMD (Naeem and Lhoták, 2008; Bodden et al., 2008). This section uses the results of our case studies with PMD for comparison with these analyses. Unfortunately, only Bodden et al. (2008) make their implementation available and even published the locations of false positives produced by their analysis; therefore, we cannot directly compare to Naeem and Lhoták (2008).

**Iterator usage.** Precision in checking the iterator protocol is comparable: Plural gives 3 false positives; Tracematches check the iterator protocol in PMD with 6 (Bodden et al., 2008) and 2 (Naeem and Lhoták, 2008) remaining warnings.

[8] The 2008 installation of the International Workshop on Specification and Verification of Component-Based Systems (SAVCBS'08) dedicated a challenge problem with several interesting solutions to this problem.

| False positives reported by Bodden et al. (2008) | 63 |
|---|---|
| Ruled out with Plural | 44 |
| Conditionally ruled out with Plural | 7 |
| Not considered | 12 |

Table 7.4: Plural's performance on false positives in checking concurrent modifications reported by Bodden et al. (2008)

The pattern that causes three false positives when using Plural— making sure a set is non-empty before retrieving one element from an iterator—also causes false positives in the Tracematch-based analyses. Naeem and Lhoták (2008) even consider them true positives and suggest that this pattern is the only source of warnings flagged by their analysis in PMD. It is worth pointing out that both Tracematch-based analyses exclude code that is never run (according to the call graph) from consideration, which may explain why Naeem and Lhoták (2008) report one less warning (2) than Plural (3).

The previously mentioned method in PMD that requires its iterator argument to be in the available state accounts for the remaining warnings reported for PMD by Bodden et al. (2008); these warnings are considered true positives by Bodden et al. (2008). Thus, our approach is more precise compared to Bodden et al. (2008) and equally precise compared to Naeem and Lhoták (2008) on the simple iterator protocol. We additionally used Plural to check that the single use of the `remove` method in PMD respects the protocol (must be called at most once per call to `next`), which was not considered with Tracematches.

**Concurrent modification.** Our approach can rule out concurrent modifications of collections and maps where the Tracematch analysis of Bodden et al. (2008) cannot. Naeem and Lhoták (2008) and Bodden et al. (2008) report very similar *numbers* of false positives for this protocol, but we do not know if the analysis proposed by Naeem and Lhoták (2008) sufferes from imprecisions in the same places as Bodden et al. (2008), since Naeem and Lhoták (2008) did not publish an implementation of their analysis or the locations where false positives.

Both Tracematch analyses report a large number of false positives in checking concurrent modifications in PMD. Naeem and Lhoták (2008) report 44 / 49 false positives. Bodden et al. (2008) report 63 / 100 false positives.[9] 51 out of their 63 false positives occur in parts of PMD considered by our case study.

Table 7.4 shows that Plural could rule out all 51 false positives reported by Bodden et al. (2008) in packages we considered. Four of these are ruled out by making assumptions about scopes; three occur in methods where Plural reports unrelated false positives.

Plural reports false positives unrelated to the ones reported with Tracematches for three reasons:

- Bodden et al. (2008) seem to better support language features that are challenging for Plural (see discussion of "sources of imprecision" in Section 7.3.2), including reflection, arrays, and static, superclass, and foreign fields.
- Tracematch analyses do not consider collections that are never iterated; we tracked all collections in our case study.

[9] Both analyses rule out unreachable code, but we do not know why the number of potential failure points differ.

- The protocols checked by Naeem and Lhoták (2008) and Bodden et al. (2008) only check concurrent modification of collections while their entries are iterated; they ignore concurrent modification of *maps* as is evident from the Tracematch protocols published for both Tracematch papers (Bodden, 2009). Our case study considered concurrent modifications of both maps and collections.

  A follow-up paper (Bodden et al., 2009) checks for *some* concurrent modifications of maps, but not others (Bodden, 2009). The later paper reports precision in terms of runtime instrumentation code that could be removed because the absence of protocol violations could be proven at compile-time. Only 12% each of instrumentation points for the patterns related to concurrent modifications of collections and maps could be removed. The remaining instrumentation causes PMD to run for more than 10 hours, compared to 13 seconds without instrumentation (Bodden et al., 2009).[10]

The bottom line is that using Bodden et al. (2008) and Plural together can rule out *all* concurrent modifications considered by both analyses. This is partially due to their different strengths: Bodden et al. (2008) support certain Java language features better than Plural currently does. They also only consider executed code and collections that are actually iterated. Plural, on the other hand, seems to be better at tracking objects through the heap, for instance through fields and nested collections, which is crucial in checking concurrent modifications in PMD. We suspect that comparison with Naeem and Lhoták (2008) yields a similar result. This suggests that our approach adds precision to existing whole-program analyses in checking challenging protocols.

## 7.4 Discussion: Lessons Learned

### 7.4.1 APIs

**Protocols are very common.** Our work with the Java standard library confirms a common research intuition: protocols are very common. Considering that even Java exceptions have a initialization protocol (whose violation can produce an exception!), the author dares to say that protocols are everywhere.

**Challenging common patterns.** There were at least three common challenges that we found across several of the APIs we specified.

1. We were surprised how prevalent *dynamic state test* methods are, and how important they are in practice. We found dynamic state test methods in JDBC, Collections, and regular expressions, and a large number of them in JDBC alone. For example, the method `hasNext` in the Java `Iterator` interface tests whether another element is *available* (cf. Section 3.1), and `isEmpty` tests whether a collection is *empty*. Since such tests are a crucial part of JDBC's facilities for executing arbitrary SQL commands, it was crucial for handling the Beehive code that our approach can express and benefit from the tests.

[10] A direct comparison with these numbers is difficult because we do not know exactly how instrumentation points are counted. As a rough approximation, there are 42 calls to `Iterator.next`, 33 calls to `Collection.add`, and 3 calls to `Map.put` in PMD rules for Java (package `pmd.rules`). These calls should represent a subset of the instrumentation points used for checking concurrent modifications with Tracematches. Plural reports 33 false positives in this part of the codebase, which means that (at least) 58% of these instrumentation points are ruled out by Plural to cause concurrent modifications in this part of the codebase.

2. We also found protocols involving multiple *interdependent objects* in these APIs (and very prevalent in JDBC). We could model these protocols by capturing and later releasing permissions.

3. We used *method cases* in JDBC and the Collections API. Method cases can be used to specify full Java iterators, which may modify the underlying collection (Bierhoff, 2006).

4. We also used *marker states* (also known as type qualifiers) in several APIs, and in particular in the Collections API (Section 7.1.2).

We believe that these are crucial to address in any practical protocol checking approach; our approach was expressive enough to handle these challenges in the APIs studied in this section. In contrast, previous approaches rarely or incompletely handle these patterns.

**Temporary borrowing.** We found several uses of aliasing which represent variations on temporarily "borrowing" an alias (permission):

1. Most methods borrow permissions for their parameters and return the permissions to the caller when they return.

2. Some API objects temporarily "capture" an alias which they only release when they are no longer used or explicitly closed. In simple cases, such as regular expressions, aliases are permanently captured in another object.

3. Aliases are temporarily borrowed "from" another object. This is in particular one way of looking at many of the "getters" that objects define for giving access to their fields: the objects temporarily lend access to the field to the caller. In other cases, that is explicitly not the intention, and instead, getters return a copy of the field to the caller. Permissions can cleanly distinguish between the two.

### 7.4.2 API Client Code

**Aliasing of API objects.** Beehive excessively aliases objects from the JDBC API in fields of multiple objects, and PMD frequently aliases Java Collection API objects from a field in views or iterators on the stack. Aliasing in PMD appears to be relatively controlled (collections are for example not referenced in multiple fields), while Beehive uses many aliases. Thus, client programs seem to make extensive use of aliasing.

**Internal protocols not enforced.** We saw a fair amount of protocols inside the client codebases we studied: Beehive expects its clients to follow a certain protcol, and PMD has a protocol for building and querying symbol tables (scopes). The similarities between these are striking:

- Neither protocol is documented, making it hard for Beehive clients or new PMD developers to follow them.

- Neither protocol is enforced at run-time; thus, if the protocol is violated, it will lead to very subtle bugs in the program. For example, changing the order in which PMD rules are run on a compilation unit could change the violations found.

- Both protocols are induced by protocols in the underlying APIs: Beehive's protocol comes from the need to close result sets once they are no longer used, and PMD's protocol avoids concurrent modifications.

**Tree structures.** Unsurprisingly, tree structures are challenging for static reasoning about programs, *if* there are parent pointers involved. While we can capture these structures with permissions (Bierhoff and Aldrich, 2008) we hope that future work can simplify working with trees automatically.

### 7.4.3 Plural Tool Usage

**Incremental benefit.** Using Plural offers incremental benefit in three ways:

1. Checking protocols for different APIs is largely orthogonal, allowing the incremental checking of more and more protocols.

2. Shallow protocols involving only one object are easier to check than more interesting protocols, as seen in the two parts of the PMD case study. This allows the programmer to move from simple to more complex checks.

3. Protocol compliance can be checked in parts of a codebase, with annotations creating cutpoints to other parts. In PMD we successfully isolated Java rules from AST and scopes, at the price of making (manually reviewable) assumptions about how the excluded parts will affect the analyzed part of the program.

**Iterative process.** We noticed the following iterative process in using the tool on existing code: we first ran Plural "out of the box" on a codebase, which results in warnings where the code calls into the APIs whose protocols the tool is checking. Adding annotations for method parameters or state invariants will "move" these errors to callers of the methods that were originally causing warnings. Adding annotations to the callers will move the warnings again until we reach the place where the API objects that these annotations track are being created. Thus, the number of annotations needed directly corresponds to how far in the call graph a API object "travels" between creation and use.

**Uniqueness or consistency.** In PMD we discovered an interesting trade-off between using unique permissions and giving objects a simple state machine:

- We tracked "report" objects (which collects the violations found by PMD rules) using unique permissions. Consequently, the `Report` class only needs a state invariant for alive (which, therefore, is comparable to a traditional class invariant in JML or Spec#) because unpacking a unique permission saves us from having to pack to an intermediate state before every method call.

- Conversely, we tracked rules using full permissions, which made it necessary to define a *quiet* state for rules. Invariants for rule fields are then associated with the *quiet* state. When rule implementations call methods they have to pack to some other state. Thus, the added aliasing flexibility of full permissions comes at the price of needed states beyond alive for specifying and verifying rules.

| Checked | Annotation rate | False positive rate | Plural runtime |
|---|---|---|---|
| 4 APIs in Beehive | 1 per 30 lines | 1 per 400 lines | 188ms per method |
| Collections in parts of PMD | 1 per 25 LOC | 1 per 188 LOC | 119ms per method |
| Iterator protocol in all of PMD | 1 per 2,600 LOC | 1 per 13,000 LOC | 62ms per method |

Table 7.5: Overhead and precision in Beehive and PMD case studies

This is consistent with our experience using Spec# (Barnett et al., 2004), whose ownership model has similarities to full permissions and guarantees class invariants to hold only when objects are in a state called "consistent".

**Annotations like types.** Throughout our case studies we used one or less than one annotation per method. This number would of course go up if every object in the program had protocols we wanted to track. (Plural ignores objects that have no protocol.) The annotations we provided, however, are in their extent very comparable to conventional Java typing information for fields and method parameters. This suggests that the overhead of tracking permissions in a program is on par with what Java developers do today to make their program compile with the Java compiler.

**Promising precision.** Plural's false positive rates (Table 7.5) represent a trade-off with the overhead of providing annotations. Removing common sources of false positives, such as inheritance and static fields, would substantially improve the false positive rates to around 1 per at least 1,000 lines of code in all case studies, which we believe to be fairly impressive considering the challenging protocols checked.

**Good-enough performance.** Plural's performance turned out to be "good enough" to be used during development. Individual methods could be checked on average in around 100ms.[11] Delays occur in practice when "choice contexts" multiply with frequent packing and unpacking (because Plural tries all possible states to pack to) and extensive use of method cases (because Plural tries all cases). In many cases, however, choices are limited quickly (e.g., due to unsatisfiable state invariants), which limits longer analysis times to a few cases.

[11] This estimate includes measurements on Nels Beckman's multi-threaded benchmark programs. Performance was measured on a Dell PC running Windows XP Service Pack 2, with a 3.2GHz Intel Pentium 4 and 2GB of RAM. Thanks to Nels Beckman for collecting performance information.

# Chapter 8

# Related Work

This section compares this research to related work. A variety of different protocol specification approaches has been proposed in the literature. Such specifications are primarily used for three different purposes, which we discuss in this order: checking programs for protocol compliance, protocol checking or simulation at runtime, and checking component compositionality at the design level. Afterwards, we discuss fractional permission inference, comprehensive program verification approaches, and protocol inference.

## 8.1 Static Protocol Analyses

This section focuses on static analyses for checking protocol compliance. We first discuss modular and then whole-program protocol checking approaches.

A plethora of approaches was proposed in the literature for checking protocol compliance. These approaches differ significantly in the way protocols are specified, including typestates (Strom and Yemini, 1986; DeLine and Fähndrich, 2001, 2004b; Kuncak et al., 2006; Fink et al., 2006; Bierhoff and Aldrich, 2007b), type qualifiers (Foster et al., 2002; Aiken et al., 2003; Chin et al., 2005a), size properties (Chin et al., 2005b), direct constraints on ordering (Igarashi and Kobayashi, 2002; Tan et al., 2003), tracematches (Bodden et al., 2008; Naeem and Lhoták, 2008), type refinements (Mandelbaum et al., 2003; Degen et al., 2007), "predicates" (Perry, 1989), and various temporal logics (e.g. Henzinger et al., 2002, and others discussed below). In these approaches, as in ours, usage rules of the API(s) of interest have to be codified by a developer.

Once usage protocols are codified, violations can be detected *statically* (like in our and most of the above approaches) or *dynamically* while the program is executing (e.g. Bierhoff and Aldrich, 2005; Dwyer et al., 2007). Many static approaches are *modular* like ours but there are also *whole-program analyses* that require no or minimal developer intervention (e.g. Foster et al., 2002; Henzinger et al., 2002). Unlike previous modular approaches, our approach does not require precise tracking of all object aliases (e.g. DeLine and Fähndrich, 2001, 2004b) or impose an ownership discipline on the heap (e.g. Barnett et al., 2004)) in order to be modular. This section discusses modular and global static protocol analyses; dynamic analyses are discussed in the next section.

Our approach complements research on helping developers accomplish their goals with the "right" framework interactions, e.g., using "design fragments" (Fairbanks, 2007): our approach allows checking

whether the concrete combination of framework interactions chosen by a developer is permitted by the framework.

### 8.1.1 Modular Analyses

Many of the proposed static approaches, including ours, are modular and require developer-provided annotations in the analyzed code in addition to codifying API usage rules. Only a few of these approaches (DeLine and Fähndrich, 2004b; Tan et al., 2003; Degen et al., 2007) target object-oriented languages like our approach.

Our approach provides a great deal of flexibility in aliasing objects, which will be discussed in more detail below. Furthermore, none of the existing modular approaches includes the concept of temporary state information (knowledge about the current state of an object that has to be "forgotten" due to possible effects through aliases).

**Checking protocol implementation code.** Ours is one of the few approaches that can reason about correctly *implementing* APIs independent from their clients. (Interestingly, all of these approaches that we are aware of are modular typestate analyses: DeLine and Fähndrich (2004b); Kuncak et al. (2006); Bierhoff and Aldrich (2007b).) Ours is the only approach (that we are aware of) that can verify correct usage *and implementation* of dynamic state test methods. Several other approaches can verify their correct usage (e.g. Mandelbaum et al., 2003; Chin et al., 2005b), but not their implementation.

We extended several ideas for checking API implementation code from Fugue (DeLine and Fähndrich, 2004b) to work with access permissions including state invariants, packing, and frames. In contrast to Fugue, subtypes are always fully substitutable with our approach and are allowed to change inherited protocols, and subclasses are free to use or not use code inherited from superclasses.

**Dealing with aliasing.** A simple mechanism to preserve soundness in modular protocol analyses is to prevent aliasing by only allowing *linear* references to stateful objects (Wadler, 1990). But programming with linear types is very inconvenient because whenever an object is passed to a function (or used otherwise), it disappears at the call site. Therefore, a variety of mechanisms to relax linearity were proposed. Uniqueness, sharing, and immutability (Aldrich et al., 2002; Boyland and Retert, 2005) have been put to use in resource usage analysis (Igarashi and Kobayashi, 2002; Chin et al., 2005b). Alias types (Crary et al., 1999; Smith et al., 2000) allow multiple variables to refer to the same object but require a linear token for object accesses that can be borrowed (Boyland and Retert, 2005) during function calls. Focusing can be used for temporary state changes of shared objects (Fähndrich and DeLine, 2002; Foster et al., 2002; Barnett et al., 2004; Morrisett et al., 2005). Adoption prevents sharing from leaking through entire object graphs (as in DeLine and Fähndrich, 2004b) and allows temporary sharing until a linear adopter is deallocated (Fähndrich and DeLine, 2002). All these techniques need to be aware of all references to an object in order to change its (or its adoptees') state.

Access permissions allow state changes even if objects are aliased from unknown places. Moreover, access permissions give fine-grained access to individual data groups (Leino, 1998). States and fractions (Boyland, 2003) let us capture alias types, borrowing, adoption, and focus with a single mechanism. Sharing of individual data groups has been proposed before (Boyland and Retert, 2005), but it has not been exploited for reasoning about object behavior. In Boyland's work (Boyland, 2003), a fractional permission means

immutability (instead of sharing) in order to ensure non-interference of permissions. We use permissions to keep state assumptions consistent but track, split, and join permissions in the same way as Boyland. Concurrently to us, Java(X) (Degen et al., 2007) proposed "activity annotations" that can be seen as full, share, and pure permissions for whole objects that can be split but not joined.

**Empirical results.** Previous modular approaches are often proven sound and shown to work for well-known examples such as file access protocols. But automated checkers are rare, and case studies with real APIs and third-party code hard to find. Notable exceptions include Vault (DeLine and Fähndrich, 2001) and Fugue (DeLine and Fähndrich, 2004b,a), which are working automated checkers that were used to check compliance to Windows kernel and .NET standard library protocols, respectively (although Vault requires rewriting the code into its own C-like language).

This dissertation shows that our approach can be used in practical development tools for enforcing real API protocols. As far as we know, this is the first work that reports on challenges and recurring patterns in specifying typestate protocols of large, real APIs. We also report overhead (in terms of annotations) and precision (in terms of false positives) in checking open-source codebases with our tool.

We suspect that empirical results are sparse because APIs such as the ones discussed in Chapter 7 would be difficult to handle with existing modular approaches due to their limitations in reasoning about aliased objects. These limitations make it difficult to specify the object dependencies we found in the JDBC, Collections, and Regular Expressions APIs in the Java standard library. Fugue, for instance, was used for checking compliance with the .NET equivalent of JDBC, but the published specification does not seem to enforce that connections remain open while "commands" (the .NET equivalent of JDBC "statements") are in use (DeLine and Fähndrich, 2004a).

Existing work on permissions recognized these challenges (Boyland, 2003; Boyland and Retert, 2005) but only supports unique and immutable permissions directly and does not track behavioral properties (such as typestates) with permissions. Similar to Fugue (DeLine and Fähndrich, 2004b), Java(X)'s "activity annotations" do not allow recovering "exclusive" access to an object once it becomes shared (Degen et al., 2007), which seems to prevent their use for encoding the iterator and JDBC protocols.

### 8.1.2 Whole-Program Analyses

Global approaches are very flexible in handling aliasing and require no or minimal developer intervention. Approaches based on abstract interpretation (e.g. Ball and Rajamani, 2001; Das et al., 2002; Hallem et al., 2002; Fink et al., 2006; Naeem and Lhoták, 2008; Bodden et al., 2008) typically verify client conformance while the protocol implementation is assumed correct. Sound approaches rely on a global aliasing analysis (Ball and Rajamani, 2001; Das et al., 2002; Fink et al., 2006; Naeem and Lhoták, 2008; Bodden et al., 2008). Likewise, most model checkers operate globally (e.g. Henzinger et al., 2002) or use assume-guarantee reasoning between coarse-grained static components (Giannakopoulou et al., 2004; Hughes and Bultan, 2007). For instance, the Magic tool checks individual C functions but has to inline user-provided state machine abstractions for library code in order to accommodate aliasing (Chaki et al., 2003). The above analyses typically run on the complete codebase once a system is fully implemented and can be very expensive. Our approach supports developers by checking the code at hand like a typechecker. Thus the benefits of our approach differ significantly from global analyses.

In contrast to modular checkers, many whole-program analyses were implemented and empirically eval-

uated. While model checkers (Henzinger et al., 2002) typically have severe limitations in scaling to larger programs, approaches based on abstract interpretations were shown to scale quite well in practice. "Sound" (see below) approaches rely on a global aliasing analysis (Ball and Rajamani, 2001; Das et al., 2002; Fink et al., 2006; Bodden et al., 2008; Naeem and Lhoták, 2008) and become imprecise when alias information becomes imprecise.

Section 7.3.1 compares our approach to state-of-the-art whole-program analyses based on tracematches (Bodden et al., 2008; Naeem and Lhoták, 2008) using one of their benchmark programs (PMD). The comparison shows that our approach matches the precision of tracematch-based analyses for checking a simple iterator usage protocol with extremely low developer overhead. Our approach could additionally rule out most false positives reported by Bodden et al. (2008) in PMD when checking the more challenging concurrent modification of collections. Another previous global typestate analysis has also been used to check the simpler iterator usage protocol, but in a different corpus of client programs and with varying precision (Fink et al., 2006).

These whole-program analyses have been used to make sure that dynamic state test methods are *called*, but not that the test actually indicated the needed state (Fink et al., 2006; Bodden et al., 2008; Naeem and Lhoták, 2008). For example, the protocols they checked require calling `hasNext` before calling `next` in iterators, but they do not check whether `hasNext` returned `true`, which with our approach is expressed and ensured easily. Tracematch-based analyses (Bodden et al., 2008; Naeem and Lhoták, 2008) currently lack the expressiveness to capture these protocols more precisely, while approaches based on must-alias information (e.g. Fink et al., 2006) should be able to, but do not in their published case studies, encode these protocols. ESP (Das et al., 2002) performs an analysis that can distinguish program branches, but at least the published example does not suggest that ESP can capture dynamic state tests. This is arguably an omission in these approaches that, given the importance of dynamic state tests in practice, we believe should be addressed.

The Metal tool (Hallem et al., 2002) found hundreds of protocol violations in Linux kernel code. DynaMine identified bug patterns related to protocol violations in Eclipse and JEdit using statistical methods (Livshits and Zimmermann, 2005). These approaches are unsound in general. Our approach verifies compliance to protocols and can therefore help prevent the defects found in these studies.

Cqual is a whole-program type qualifier inference system for C that, like Metal, found locking protocol violations in the Linux kernel (Foster et al., 2002). In many cases, the placement of "focus" (or "restrict") to temporarily change the state of an object can also be inferred (Aiken et al., 2003). While Cqual allows type qualifiers to change over time (because it is flow-sensitive) it that type qualifiers cannot encode protocols involving multiple objects, such as the ones found in JDBC.

We do not claim to comprehensively compare our approach's precision relative to whole-program analyses. But we do point out that our approach, unlike whole-program analyses, can reason about API implementations separately from clients and handles dynamic state tests soundly, as discussed above. Reasoning about API implementations separately from clients is critical for libraries such as Beehive that may have many clients. Our approach also seems to be superior to state-of-the-art tracematch-based analyses in checking "deep" protocols in challenging codebases.

## 8.2 Protocols at Runtime

Protocols can also be used to check or simulate running systems. Statecharts (Harel, 1987) are used to visually specify reactive systems with hierarchical state machines. Statecharts can be used to generate executable simulations of the specified system (Harel and Naamad, 1996). Butkevich et al. (2000) use regular expressions to describe protocols (that are translated into labeled transition systems). Runtime checks are used to ensure protocol compliance of a running system. We previously implemented a runtime protocol checker for typestate protocols specified with union and intersection types (Bierhoff and Aldrich, 2005). Dwyer et al. (2007) shows how to reduce the overhead of such checks and recent tracematch-based analyses have been used to remove runtime checks where possible (Bodden et al., 2008, 2009).

This dissertation proposes a static protocol verification approach that does not require a runtime protocol checker. No explicit runtime representation of access permissions is needed. However, in order for verification to succeed, runtime tests in the form of `if` statements may be required. We employ mechanisms similar to Statecharts' AND- and OR-states to define state spaces. Our method specifications involve multiple objects and are therefore quite different from state transitions in Statecharts.

## 8.3 Verifying Component Compositions

This section summarizes approaches for checking whether protocols of independently developed components are compatible. The architecture description language Wright (Allen and Garlan, 1997) includes protocol specifications based on CSP (Hoare, 1985). A model checker for CSP is used to verify compositionality, i.e., verify that components are connected in such a way that they do not deadlock. FSP (Magee et al., 1999) specifies component behavior with labeled transition systems (LTSs), and compositions of LTSs can be checked for compositionality as well. This check adds flexibility to assume-guarantee reasoning (Giannakopoulou et al., 2004). Interface automata (de Alfaro and Henzinger, 2001) describe protocols as state machines that can likewise be checked for compositionality.

These approaches provide design-level reasoning about component compositions. Our verification approach essentially checks compositionality as well, but it does so by checking the actual code that uses an API for protocol compliance.

## 8.4 Fractional Permission Inference

Fractional permissions were proposed by Boyland for avoiding data races in concurrent programs (Boyland, 2003) and have since been used in a variety of contexts. In particular, we introduce new kinds of permissions and the notion of state guarantees; we use linear logic to compose predicates from permissions; and we track typestates, which Boyland does not consider.

This dissertation provides a permission inference system for a fragment of linear logic predicates that is polymorphic with respect to fractions and describes its implementation in a protocol checking tool for Java that can fully check individual methods in 100ms on average.

Terauchi and Aiken (2008) previously proposed a fraction inference system for checking data races in concurrent programs whose implementation is based on a linear programming engine (Terauchi, 2008). In their work, control flow merges introduce new fraction variables that are constrained to be smaller than the incoming fractions. Function signatures are also inferred with fraction variables. Constraints can then be

collected by scanning the entire program once, even in the presence of conditionals and loops, allowing constraint collection to be linear in the size of the program. They also used fractions to infer legal orderings of side-effecting operations based on "witnesses" (Terauchi and Aiken, 2005).

Conversely, we collect constraints with a intra-procedural flow analysis, which is polynomial in the worst case (Nielson et al., 1999, Chapter 6) and requires annotations for permissions passed into and out of methods. An advantage of our modular approach is that permissions can be consumed in loops, which was not previously possible (see Terauchi, 2008, Section 5). We are also able to use universal quantification in function signatures (currently provided with explicit annotations), which may be more flexible than inferring concrete fractions for signatures. These differences are due to our support for polymorphic fractions, which are not available in previous work.

Ongoing work in Boyland's group on avoiding data races unfortunately elides the details of their fraction inference implementation (Zhao, 2007). Bornat et al. (2005) combined Boyland's fractional permissions with separation logic, but their approach is intended for hand-written proofs about program correctness. VeriFast is an automated proof checker for separation logic (Jacobs and Piessens, 2008) that to our knowledge supports Boyland's fractional permissions, but the details of the implementation are again unknown. VeriFast seems to require significant developer input.

Other ongoing work encodes Boyland's fractions using integers representing percentages between 0 and 100 as well as "infinitesimal" fractions in first-order logic (Leino and Müller, 2009). Loop invariants have to be provided by hand, while our implementation infers loop invariants. Unlike with polymorphic fractions, it appears that this unusual encoding will require developers to use concrete percentage values or concrete numbers of infinitesimal fractions in most of their method pre-conditions and loop invariants. Infinitesimal fractions come in a duplicable variety, but those cannot be used to borrow or otherwise temporarily alias objects, such as database connections.

## 8.5 Comprehensive Program Verification

Comprehensive program verification approaches can also be used to reason about protocols. These approaches employ logical predicates in specifications. We first discuss approaches based on classical first-order logic and then consider separation logic.

**First-order logic.** Early approaches to comprehensive program verification include Alphard (Wulf et al., 1976) and later Larch (Guttag et al., 1985). In both approaches, datatypes are specified with invariant predicates, and their operations are specified with pre- and post-condition predicates. Alphard uses uniqueness for controlling aliasing and uses "states" with invariants for more concise description of datatype implementations. Eiffel (Meyer, 1992) pioneered the idea of executable pre- and post-conditions. Methods in Eiffel declare their pre- and post-conditions using Eiffel code and check them dynamically.

The JML is a behavioral specification language for Java that incorporates ideas such as datatype invariants from Larch and executable specifications from Eiffel (Leavens et al., 1999). The JML (Leavens et al., 1999) is very rich and complex in its specification features; it is more capable than our system to express object behavior (not just protocols), but also potentially more difficult to use due to its complexity. Verifying JML specifications is undecidable in the general case. Tools like ESC/Java (Flanagan et al., 2002) can partially check JML specifications but are unsound because they do not have a sound methodology for handling aliasing. Spec# is comparable in its complexity to the JML and imposes similar overhead. The

Boogie methodology allows sound verification of Spec# specifications but requires programs to follow an ownership discipline (Barnett et al., 2004).

Our system is much simpler than these approaches, focusing as it does on protocols, and can be automated to a higher degree as a result. Our treatment of aliasing makes our system sound, where ESC/Java is not. While the treatment of aliasing in our system does involve complexity, it gives the programmer more flexibility than Boogie's ownership methodology while remaining modular and sound. Because it is designed for protocol verification in particular, our system will generally impose smaller specification overhead than the JML or Spec#. Additionally, ownership-based approaches cannot express dependencies of multiple objects on a single "server" object, while permissions allowed us to express these patterns for three major Java APIs.

Crucial for reasoning about object-oriented programs is the treatment of subtyping and inheritance. Behavioral subtyping formalizes rules for checking whether subtypes conform to the behavior of supertypes (Liskov and Wing, 1994). The JML and Spec# guarantee behavioral subtyping through specification intersection (Dhara and Leavens, 1996). The JML has no particular methodology for handling inheritance. Spec#'s handling of inheritance is based on Fugue's frames (DeLine and Fähndrich, 2004b). Our approach includes a logic-based check for behavioral subtyping of protocol specifications. Like Fugue and Spec#, we handle inheritance using frames. Our approach of unpacking one frame at a time differs from Fugue but is similar to Spec#'s "local expose". Unlike in Fugue, the states of different frames can be decoupled, and method overriding requirements are relaxed in our approach.

**Separation logic.** Separation logic extends classical logic to simplify reasoning about shared mutable state (Reynolds, 2002). Its operators are very similar to those defined in the MALL fragment of linear logic (Girard, 1987) and can be combined with traditional classical logic operators. Separation logic is undecidable in the general case, but heap shapes have been successfully inferred automatically based on separation logic (Magill et al., 2006; Calcagno et al., 2009).

Separation logic completely isolates effects on separate parts of the heap as `unique` permissions do. Other permissions are not available directly in separation logic, but separation logic predicates can be used to pass shared objects around between different parts of the program. Immutability can be natively included by combining separation logic with fractions (Bornat et al., 2005). Parkinson and Bierman (2008) proposed "predicate families" for using separation logic in the context of object-oriented programming. Predicate families have many similarities to our state invariants and consequently provide comparable flexibility to subclasses in using inherited code.

## 8.6 Protocol Inference

Recent static API protocol inference systems for C (Henzinger et al., 2005) and Java (Nanda et al., 2005) use typestates as their protocol abstraction. Remarkably, the inferred Java protocols (Nanda et al., 2005) are very similar to what we can enforce. Protocols can also be "mined" from revision histories (Livshits and Zimmermann, 2005), typically by employing statistical methods to identify "common patterns". These approaches are complimentary to ours in that the inferred protocols could be specified and checked using Plural, which could reduce the annotation burden of specifying API protocols.

# Chapter 9

# Conclusions

## 9.1 Validation of Hypotheses

This dissertation set out to demonstrate that typestate-based protocols with access permissions can enforce API protocols in practical object-oriented software. This thesis is not directly testable; therefore, we have focused on four testable hypotheses that aim at establishing our approach's *well-foundedness* and its *applicability* to common software practices, in particular with respect to APIs, conventional programming languages, and how code is written in practice.

Our first hypothesis is that our approach can succinctly capture common protocols; our second hypothesis is that the approach can soundly and modularly check protocols in object-oriented code. Our third hypothesis is that the approach can be sufficiently automated for use by software developers, and our fourth hypothesis is that our approach can be used to check off-the-shelf software.

Notice that our thesis does *not* claim that our approach works well with all APIs and all code written to date. Instead, we have attempted to "simulate" our approach's use in practice by considering a handful of real, commonly used APIs as well as third-party open-source software.

The following sections discuss the evidence we gathered in support of each of our hypotheses.

### 9.1.1 Capture Common Protocols Succinctly

Our first hypothesis establishes the applicability of our approach for API protocols occurring in practice.

> Typestate-based specifications with access permissions can succinctly capture API protocols commonly occurring in practice.

We support this hypothesis with evidence from case studies on several Java APIs in wide practical use (Section 7.1). We identify a number of challenging recurring patterns, all of which can be sufficiently expressed with our approach. Furthermore, the number of annotations needed for capturing protocols with our approach is small, especially when compared to informal documentation, and can be provided quickly.

The studied Java APIs are all part of the Java standard library (that is included with Java 6) and include Java Collections, Java Database Connectivity (JDBC), Regular Expressions, Exceptions, and Character Streams. According to Google searches, all of these were in wide practical use in 2008 (with Regular Expressions being least frequently used), and we have no indication that this situation would change.

With roughly 450 methods, JDBC was the most complex API we studied. We specified protocols for JDBC with about 2 annotations per API method. Each of these methods are documented with approximately 20 lines of English text in the Java standard library. The author could provide the protocol-related annotations in a short amount of time, specifying around 100 methods per day, which was largely spent reading the extensive documentation included with the JDBC API.

We found three patterns to be recurring in at least 3 of the 5 APIs we studied: *dynamic state tests*, *dependent objects*, and *method cases*. These have limited support in existing protocol checking approaches but could be handled by our approach in the examples we studied (Section 7.4).

### 9.1.2 Sound Modular Checking

Our second hypothesis characterizes the theoretical properties of our approach.

> Typestate-based protocols with access permissions can be verified in a sound modular fashion.

This hypothesis is supported by a formalization of our approach as a type system (Chapter 4) and a proof of soundness (Bierhoff and Aldrich, 2007a). The proof of soundness, informally speaking, guarantees that programs that typecheck in our type system will never violate declared protocols at run-time.

The formalization demonstrates a number of properties that we consider important for its practicality:

- *Soundness*, i.e., a guarantee that the approach does not miss any potential protocol violations, establishes the approach's well-foundedness.
- *Modularity*, i.e., the ability to check parts of a program separately from the rest, is crucial for scaling to large codebases, compositional reasoning about programs developed by many developers, and reasoning about libraries (or application "layers") separately from their clients.
- *Refinement typing* means that we can check protocols on top of a conventional Java-like type system, without changing the execution semantics of the underlying language.
- *Aliasing flexibility*, i.e., the ability to reason about the protocol of objects that are referenced from multiple places in the program (modularly). This is the most visible distinguishing feature of our approach and provides us with crucial flexibility for handling aliasing patterns we found inherent in APIs and in the open-source software we studied.
- *Support for dynamic tests* allows developers to deal with uncertainties inherent in many APIs and allows overcoming imprecisions in our type system with conventional `if`-tests.
- *Flexible overriding* allows overriding methods *without* calling the overridden method, which is commonly done in practice.

These properties characterize our approach as potentially useful; the remaining hypotheses establish its applicability in practice.

### 9.1.3 Automation

We were concerned that our approach could only be used for manually proving programs correct with respect to protocols (with hand-written proofs such as the ones shown in Chapter 3). Our third hypothesis emphasizes that our approach is amenable to automated reasoning with little developer intervention.

> Access permission-based protocol checking can be automated with annotation burden comparable to conventional type declarations and used with conventional programming languages.

We describe a type inference system inspired by constraint logic programming that reduces permission tracking in the type system to linear constraints (Chapter 5). These constraints can be checked automatically for satisfiability.

We embedded this inference algorithm into a tool, Plural, for checking protocols in the Java programming language (Chapter 6). The tool requires developer annotations for method parameters and fields but processes individual Java methods fully automatically.

Measurements show that developer-provided annotations are in their extent comparable to conventional Java typing information (both for describing API protocols and for tracking objects from APIs in client code). In our case studies we used about 1 annotation per method of API client code and 2 annotations per method for specifying the API itself. These annotations often have the flavor of conventional types in that they "borrow" a permission for the method receiver or a method parameter. Borrowing means that the permission that is required by a method is returned to the caller when the method returns. In other cases, permissions are "consumed" by methods, i.e., required but not returned. These are very simple patterns, and our annotations can express them efficiently, making them visually about as "large" as the conventional Java types that have to be provided for every method parameter in Java.

Measurements also show that the prototype tool can check protocols fast enough for interactive use. On average, the tool checks individual methods in around 100ms; the biggest method we encountered in our case studies was over 500 lines long.

### 9.1.4 Practical Checking

Our final hypothesis claims applicability of our approach to software found in practice.

> The approach can enforce API protocols in off-the-shelf object-oriented software.

We used our tool for checking API protocols in two open-source codebases, namely Apache Beehive (Section 7.2) and PMD (Section 7.3.2). Both are third-party codebases that extensively use Java APIs we specified, in particular JDBC (Beehive) and Java Collections (PMD).

2,000 lines of Beehive code could be checked by our tool with 5 false positives and while finding 2 potential problems in the code (that do not, to our knowledge, manifest in runtime errors). All but one call into JDBC (due to an invariant we could not express) could be checked. Three usages of a Java Collection through a static field and one reflection-related warning account for the remaining false positives. Beehive is itself intended to be used as a library in a larger application, and we could use our tool to define a protocol for using Beehive and establishing a correspondence between Beehive's own protocol and the protocols of APIs that the Beehive code uses.

In PMD, a program of close to 40 KLOC, we could successfully rule out 98% of potential violations of the Java Iterator protocol (which is part of the Collections API). This required only 15 annotations, which could be provided by the author in 75 minutes.

We also checked 8.7 KLOC of PMD for concurrent modifications of collections while collections are iterated, using annotations to make assumptions about the rest of the program. The close to 600 annotations required could be provided in 18 hours. 46 false positives remain in this part of PMD, which are discussed in detail (Section 7.3.2). Many of these false positives could be avoided; the majority arises from an inheritance pattern that our type system does not support, but we believe that our approach could be modified to include this pattern.

PMD has also been the subject of case studies with state-of-the-art whole-program protocol analyses based on *tracematches* (Naeem and Lhoták, 2008; Bodden et al., 2008). Our approach successfully rules out all false positives reported by Bodden et al. (2008) related to concurrent modification of iterated collections in the part of PMD we considered (Table 7.4). Precision of the two approaches for checking compliance to the simpler iterator protocol is comparable. This suggests that "shallow" protocols can be checked relatively easily with both approaches; "deep" protocols can cause problems for whole-program tracematch analyses and can be checked more precisely with our approach at the price of having to annotate the program.

In certain cases, refactorings enable checking protocols automatically in code that originally could not be analyzed. Most of these refactorings are to avoid shortcomings in our tool implementation such as a relatively simplistic local alias analysis.

### 9.1.5 Thesis

The thesis of this dissertation is:

> *Typestate-based protocols with state refinement and access permissions can be used for automated, static, modular enforcement of API protocols in practical object-oriented software.*

This dissertation has provided evidence for our approach's *well-foundedness* and its *applicability* to common software practices, in particular with respect to APIs, code, and how code is written. We found numerous opportunities for improving our tooling, and motivation to improve our approach's handling of inheritance. On the other hand, our approach of using access permissions to reason about typestate in the presence of aliasing has largely held up to practical demands. In particular, the improved aliasing flexibility has allowed us to capture and enforce API protocols that could not be captured with previous modular program verification approaches. One of our case studies suggests that our approach can check protocol compliance in many situations that cause imprecisions in state-of-the-art whole-program protocol analyses (Naeem and Lhoták, 2008; Bodden et al., 2008).

## 9.2 Concerns

Despite the evidence provided in this dissertation, a number of concerns remain regarding the potential success of this research. This section discusses the following concerns:

- Differences between programs, programmers, and programming languages.

- Applicability to frameworks.
- Expression cost.
- Adoptability.
- Tool performance.
- Concurrency.

### 9.2.1 Variations in Development Practices

One possible objection to this work could be that it may not be applicable to all programs, all programmers, or all programming languages.

- *Program variability.* The variations between programs can be large, and it is impossible for us to enumerate all possible programs and evaluate how our approach might work on them. But our case studies did achieve considerable breadth in covering common APIs and their protocol patterns, suggesting limited variation there. This suggests that our approach can express many kinds of API interaction patterns occurring in practical programs.

- *Programmer differences.* Programmers have different approaches to how they solve problems. We expect all programmers new to a particular API to benefit from our approach. Experienced programmers seeking confirmation that their work is correct will likewise benefit from our approach directly. Experienced programmers interested in the quickest possible solution may benefit from our approach by saving testing time and pointing out problems on rarely exercised code paths. Less desirably, developers may choose to ignore protocol violation warnings, unlike conventional typing errors, since they do not prevent the program from being run.

- *Programming language dependencies.* The approach presented in this dissertation was designed for conventional object-oriented languages. While our tool was built specifically for Java, we see little problem in using the approach with C# or C++ (with additional support for function pointers). Imperative languages without subtyping (such as C) could also benefit from this approach. The soundness of our approach depends on type safety, but it will work with weakly typed languages such as C and C++. Dynamically typed languages (such as SmallTalk, Python or JavaScript) would require more significant adaptation.

  Most functional languages support shared mutable state, but common belief is that it is rarely used. However, functional programs interact with the world using elementary libraries that give for instance access to files, the screen, or databases. These interactions follow similar protocols to the ones defined in the Java standard library, and our approach can be used to enforce them. Furthermore, our case studies show that protocols of elementary APIs are not limited to the fringes of programs but affect higher-level code as well, suggesting that our approach could significantly help functional programmers in following protocols.

### 9.2.2 Applicability to Frameworks

Software frameworks have made significant inroads in the development of GUI-based desktop and Web applications, among other domains. Frameworks provide an application skeleton to which application developers can add "plug-ins" (extension code) that interacts with the framework code to enhance or change the framework's default functionality. Frameworks typically require plug-ins to implement certain "lifecycle methods" that the framework invokes at well-defined moments. From a plug-in's perspective, a framework defines—often overwhelmingly many—APIs that the plug-in can use to *call back* into the framework during the execution of a plug-in lifecycle method (Fairbanks, 2007); plug-ins act as API clients in this regard. In order to allow this interplay, frameworks and plug-ins share objects (through which lifecycle methods can be invoked and callbacks into the frameworks can be performed).

The protocols for framework-plug-in interactions are often intricate, and it appears that many of them could be formalized and enforced using typestates (Jaspan and Aldrich, 2009). In this sense our approach therefore complements recent research on helping developers accomplish their goals with the "right" framework interactions, e.g., using "design fragments" (Fairbanks, 2007): our approach allows checking whether the concrete combination of framework interactions chosen by a developer is permitted by the framework. Furthermore, this may raise the awareness of framework developers for the protocols they impose, leading to potentially cleaner framework designs that what we see in practice today (Fairbanks, 2007).

### 9.2.3 Expression Cost

Our case studies show that annotations for using our tool have the same extent as conventional typing information when tracking every object in a program. This is a significant cost that, as our case studies show, can be hard to retro-fit into existing code. The case studies also indicate that our approach can be incrementally phased into existing code, by starting with one simple API protocol in a part of the codebase and expanding out from there. Given the similarity to conventional typing, it appears that annotations can be realistically provided from the start in newly written software, but this intuition should be confirmed in future research.

### 9.2.4 Adoptability

The approach presented in this dissertation proposes of a number of non-trivial concepts, including 5 different kinds of permissions, state refinement, state guarantees, a distinction between "frame" and "virtual" permissions, and the use of "state invariants". Developers cannot be assumed to be familiar with these concepts, creating challenges for adopting this work into practice.

Many of our concepts are designed to mirror intuitive notions used by developers today. For instance, we have evidence from the Java Streams library that the developers of this library documented state invariants informally (Bierhoff and Aldrich, 2005). Additionally, the concept of state refinement is already familiar to many developers from Statecharts as they are part of the UML (Rumbaugh et al., 2004). But the question remains how easy it is to understand these concepts in the first place.

We believe that the biggest challenge in explaining this work to others is the systematic distinction between read-only and read-write access (which is pervasive in the approach), followed by the distinction between frame and virtual permissions (which is relevant for subclassing). While treating read-only access

special is standard in program verification (Barnett et al., 2004) and functional programming, it appears to be largely implicit in conventional imperative and object-oriented programming practices.

We could retrofit permissions quite well into existing object-oriented programs. Moreover, collaborators at Carnegie Mellon University have adopted our approach for their own research (Beckman et al., 2008, and others), and the approach has been used in the classroom and in the AcmeStudio project at Carnegie Mellon. This suggests that the approach is compatible with implicit programming notions and accessible at this point to well-qualified developers. Part of these successes, we believe, is due to our efforts to largely shield developers from the complexities of using linear logic directly. In the future, we hope to further simplify our model for application-specific purposes. We also plan to investigate adoptability challenges more thoroughly in user and field studies.

### 9.2.5 Tool Performance

We noticed that subjective delays in running the tool can occur in two situations: extensive unpacking and packing due to field accesses interspersed with method calls, and extensive use of method cases. Both situations permit many possible choices (such as, which state to pack to, or which method case to choose). If these choices are not determined by the program then the tool has to carry them forward, which can lead to snowball effects of choices multiplying.

We do not think that these issues are a threat to the approach's practicality: we see them only in rare cases, they only occur in severely underspecified programs, and they are partially a result of our tool not being optimized for these situations. Furthermore, these are problems *with the tool* that are largely due to our desire to analyze methods *fully automatically*; simple programmer-provided information could eliminate these problems by telling the tool which choice to make (which state to pack to, or which method case to choose). In fact, we believe that such annotations would capture valuable programmer intent and could easily become part of future tools or programming languages based on our approach.

Our tool checks constraint satisfiability based on Fourier-Motzkin elimination, which creates an exponential number of formulas in the worst case when eliminating variables. We experience good performance in practice, which seems to result from a carefully chosen order of eliminating variables (Section 5.2.2).

The alternating quantification in constraint formulas prevents us from using linear programming for checking constraint satisfiability, as was done in previous work (Terauchi, 2008). Alternating quantification is a well-known challenge for SMT (Satisfiability Modulo Theories) solvers. Other researchers have successfully used Farkas' lemma to remove alternating quantification (Gulwani and Tiwari, 2008; Gulwani et al., 2008). This would enable the use of linear programming tools at the price of being incomplete (Gulwani et al., 2008).

### 9.2.6 Concurrency

This work does not enforce correct synchronization when accessing shared data in concurrent programs: our analysis assumes single-threaded execution or at least a correctly synchronized multi-threaded program. Nels Beckman has proven a variant of our approach sound for concurrent programs with Atomic blocks (Beckman et al., 2008) and extended the Plural tool to such programs in his tool, NIMBY[1]. A key insight of his work is to use aliasing (as encoded with permissions) as a sound approximation of thread sharing, which

[1] NIMBY is available together with Plural at `http://code.google.com/p/pluralism/`.

*forces* the correct placement of Atomic blocks where thread-shared data is accessed or interleavings with other threads would invalidate state information assumed by the current thread.

## 9.3 Discussion

We made a number of observations while investigating our approach in this dissertation that we discuss in this section, including the possible pay-off from using our approach, its most important features in practice, advice we can give to API designers, and the effects we believe this work can have on software engineering practice.

### 9.3.1 Pay-off and Incrementality

Since many of the problems our approach catches will be found during testing, the added benefit of applying our approach after the fact to deployed code is small unless it is vital that the code being checked is correct.

Much greater benefit comes from using the approach during development and maintenance tasks, which our modular approach permits easily. There are several situations in which we expect the benefit to be particularly large:

- *Gaining familiarity with a API.* Rather than having to extensively read API documentation (if even available) developers trying to use an API they are not familiar with will be able to "just start using" the API, and our approach will tell developers when they violate protocols associated with the API. For complex APIs this strategy can be significantly faster, in particular when important protocol rules are documented in unexpected places.

- *Maintenance and agility.* Any code change, especially by a developer that did not originally write the code being changed, may violate existing protocols. Our approach avoids the overhead of extensively re-testing the program after every code change. This also makes our approach attractive for use in agile software development, despite the extra time it may take developers to provide needed annotations.

- *Intermediary library implementation.* Our approach seems to lend itself to establishing consistency between a library's protocol and the protocols of APIs used by the library, as we saw with Beehive.

- *Complexity.* Obviously, it is harder to follow complex protocols, in particular those involving multiple objects, than straightforward protocols. Our approach seems to be able to capture and enforce such protocols.

Additionally, our approach seems to support graceful ways into using it: checking protocols of different APIs in a given codebase is largely orthogonal (as we saw with Beehive), simpler protocols are more easily checked (as we saw with PMD), and protocols can be checked in only a part of the code (as we also saw with PMD).

### 9.3.2 Essential Features

While we designed our approach to be as flexible as possible, there are certain features that we found essential for modeling and enforcing protocols in practice:

- Borrowing and capturing objects (Section 6.4.2). Methods (and constructors) seem to either borrow or capture permissions to parameters. This means that methods return either the same permission that they require, or no permission at all, although other situations are certainly conceivable. When permissions are captured they often end up in invariants of another object, which we discuss below. Plural provides annotations like `Full` that capture these two cases conveniently.

- State dimensions and state refinement (Section 3.1). We found our hierarchical notion of state machines convenient for modeling protocols, and they seem to make specifications more concise, which we believe is important for adoptability in practice.

- Support for dynamic state tests (Section 6.3.6). Dynamic state tests are ubiquitously used in API protocols, and support for them is essential for ensuring correct protocol usage.

- State guarantees (Section 3.2) and object dependencies in general. API protocols in practice seem to go beyond simple state machines for individual objects and include multiple interrelated objects. Permissions let us model these dependencies and we think that this flexibility will be crucial in modeling many API protocols, even in the context of more complete behavioral models such as first-order logic. The idea of state guarantees was crucial for expressing dependencies in JDBC in particular, and it will be interesting whether state dependencies can be generalized beyond finite state machines.

- Capturing permissions for the lifetime of another object or reference (Section 7.1.2). Capturing objects for the lifetime of another object let us express dependencies between objects in APIs (see above). In another sense, this can be seen as lending an object to another object (instead of a method, as above). We can also borrow permissions "from" another object (Boyland et al., 2007) for the lifetime of the reference holding the borrowed object, by leaving the borrowing object unpacked for that time.

- Local views on objects. Permissions allow us to talk about what we know about an object from the perspective of a given method. They encode what other references may do without explicit knowledge of where these references may be. This facilitates compositional automated reasoning. It also seems to line up well with a programmer's way of thinking about a given reference from the perspective of the code module she works on.

- Distinguishing read-only access through a particular reference. This feature of permissions represents important design intent that, for instance, allows us to talk about whether or not an iterator's `hasNext` implementation is allowed to modify the iterator (Section 7.2.2). It also permits "strong updates" in the presence of aliasing (with full permissions), which is an improvement over linearity-based approaches.

### 9.3.3 Advice to API Designers

From our case studies on Java APIs we extract a number of lessons for API designers based on patterns that were hard or awkward to express with our approach.

- *Avoid methods with multiple purposes.* Important design intent is lost when methods implicitly have multiple outcomes. For instance, there is only one method for creating an iterator over a collection,

and it is not clear whether this iterator will modify the iterated collection or not (Bierhoff, 2006). While method cases can elegantly handle such methods, we believe their behavior is not easily predictable from code that uses them.

- *Document "mode" restrictions on methods with these methods.* In our work with JDBC's `ResultSet` interface we discovered several dozen methods into the case study that there were two "modes" to be distinguished that affected previously specified methods (Section 7.1.1). However, the affected methods' documentation did not indicate this mode dependency. Such restrictions should be documented with all methods that are affected.

  Furthermore, if modes exist then methods should not just be marked as "possibly unavailable". In the Java Collections API, many methods are marked as "optional" without specifying exactly under which circumstances the methods will work. It turns out that these methods are available for "modifiable" collections (Section 7.1.2), and we believe it would be helpful to state this in the documentation.

- *Avoid subtypes that do not fully follow inherited protocols.* In the Java Streams API it appears that "pipes" have a more restrictive protocol that any old stream: while other streams can be closed at any time, sinks (instances of `PipedInputStream`) can only be closed when the end of the stream was detected (Section 3.2), violating behavioral subtyping (Liskov and Wing, 1994). This problem can be avoided with a more restrictive specification of the basic stream protocol and being more permissive in subtypes such as `FileInputStream` that can definitely be closed at any time. But problems remain when such streams are, for instance, wrapped into a `BufferedInputStream`. We believe that behavioral subtyping should be respected in APIs because using abstract supertypes defined in APIs can otherwise lead to runtime errors that are hard to understand.

### 9.3.4 Effect on Software Engineering Practice

We hope that this work will have the following effects when adopted into software engineering practice:

- *Cleaner APIs.* Specifying permission-based protocols for APIs should make these APIs cleaner because awkward protocols or aliasing will become apparent.

- *Confident aliasing.* Developers will be able to more confidently use aliasing in their programs because an automated tool helps them predict the non-local effects of aliasing. We believe that this has significant potential for easing software construction.

- *Rapid API use.* Tool support will allow rapid *and* correct use of unfamiliar APIs, facilitating collaboration (defined broadly). This could increase the number of APIs being used in a project, or simply accelerate software construction.

- *Facilitation of reuse.* This work makes it easier to make a code module reusable as an API. It will be easier to document how to use the module. Protection against protocol violations is either unnecessary (in trusted codebases) or could potentially be automatically generated.

- *Dependability.* Static assurance of protocol compliance will lead to increased dependability both of the client side and the provider side of APIs.

## 9.4 Contributions

**Novel abstractions: State refinement and access permissions.** New abstractions are proposed for protocol specification and enforcement based on typestates that

- improve expressiveness, handling of subtyping and inheritance, and reasoning under aliasing compared to previous typestate-based approaches.
- can be used with existing object-oriented programming languages.

These abstractions allow more succinct protocol specifications (by avoiding state explosion problems and conveniently encoding uncertainty) and support important API protocol patterns that are insufficiently supported in previous protocol checking approaches.

**Sound modular protocol checking under aliasing.** Protocol checking based on access permissions is formalized as a modular type system. A fragment of this system was proven sound (Bierhoff and Aldrich, 2007a). Due to the use of access permissions, protocol compliance can be checked in a modular fashion even if objects can be manipulated though aliases from elsewhere in the program.

**Inference.** An inference system for syntax-directed tracking of polymorphic access permissions is presented. The inference system combines resource tracking techniques from linear logic programming with constraint logic programming to reduce protocol checking to quantified linear constraints, which can be checked for satisfiability using Fourier-Motzkin elimination.

**Practical tooling.** A working prototype tool demonstrates that the approach can be used in practical automated software development tools for conventional programming languages. Performance measures indicate the tool to be suitable for interactive use. A number of extensions to the proposed approach were implemented in the tool that make it more useful in practice.

**Evaluation.** Case studies evaluate the approach's ability to capture and enforce commonly used API protocols in third-party open-source software. Developer-provided design intent is in its extent comparable to conventional types, and recurring API patterns can be successfully captured.

## 9.5 Future Work

There are many avenues for future research directly related to this work.

First, one could apply this work specifically to a number of well-known problems including the following, all of which have received significant attention on their own:

- Object initialization protocols (Fähndrich and Xia, 2007; Qi and Myers, 2009) are a special case of protocols.

- Static avoidance of null dereference errors (Fähndrich and Xia, 2007) and buffer overruns (Hackett et al., 2006) may be increased in cases where pointer properties are state-dependent. For instance, fields may have to be non-null in certain states, which we can encode with Plural.

- Memory corruption (through pre-mature memory deallocation) and memory leaks (through missing memory deallocation) in programming languages with explicit memory deallocation can be avoided by ensuring that a `unique` permission is available for deallocating memory (which prevents corruption), and that `unique` permissions are not silently forgotten (which prevents leakage in many cases).

- Information flow tracking could be encoded with "marker states". For instance, we may associate data with a "secret" marker state (type qualifier) and make sure that secret or possibly-secret data does not become public.

- SQL injection could be statically prevented with the following classification of user data as "safe" or "harmful", which accounts for dynamic checking that user data is in fact harmless:

```
@States(value = {"safe", "harmful"}, dim = "dangerous", marker = true)
public interface UserData {
    @Pure("dangerous")
    @TrueIndicates("safe")
    boolean isSafe();

    @Imm
    @ResultUnique(ensures = "safe") UserData makeSafe();
}
```

- Implementations of security protocols could be checked for compliance with the desired protocol. This is in particular useful for avoiding easy-to-miss loopholes such as omitted runtime checks in "secure" clients. For instance, the Kerberos protocol requires clients to generate fresh tokens that are later compared to answers from the Kerberos server (Neuman and Ts'o, 1994). If these tokens are not in fact fresh, or are not compared later, then the client becomes vulnerable to replay attacks. This work can be used to enforce the comparison, and it can put restrictions on what data is considered fresh (for instance, constants could be forbidden).

Second, we hope to develop techniques for further reducing the annotation burden of using Plural. One promising way of doing so could be to turn our dataflow analysis for tracking permissions into an interprocedural analysis which, for instance, could check an entire compilation unit or larger code module fully automatically.

Third, our approach promises to allow performance improvements: it can detect unnecessary runtime tests in programs, which could be automatically removed by a compiler. This may in particular simplify library implementations, which will need to perform fewer runtime checks to ensure that clients are using the library according to its protocols. Preliminary measurements show that iterators over `java.util.ArrayList`, an array-based list implementation in the Java Collections API, speed up between 14% and 20% (depending on the length of the list, $p < 0.01$ over 20 runs) if they do not perform defensive runtime checks. On the other hand, if security is an issue, we may want to *generate* such runtime checks to prevent possible exploits. An interesting research question is which checks in current library implementations are needed for security reasons and which ones are not.

Finally, the time has come to perform user and field studies to investigate the effects of this work on software development and maintenance. In this context it will become crucial to make protocol specifications for APIs available to interested users. Protocol mining techniques could be used to synthesize typestate protocols, and subsequent static checking results could improve the accuracy of these protocols.

In addition, this research may inform other lines of research.

First, this approach can be used to aid engineers in developing concurrent software. Nels Beckman is successfully employing permissions to ensure race freedom in concurrent programs that use Atomic blocks for mutual exclusion (Beckman et al., 2008). Even if no protocols are tracked, permissions can still enforce the consistent use of Atomic blocks (or, potentially, locks) when accessing shared mutable data, which we think is a very exciting and promising result.

Second, a very exciting avenue for future research is the use of permissions in program verification. The JML and Spec#, two leading approaches in this space, use ownership to enable sound modular program verification (Barnett et al., 2004; Dietl and Müller, 2005). It appears that permissions can track first-order logic predicates just like typestates (Bornat et al., 2005; Bierhoff and Aldrich, 2008; Leino and Müller, 2009). This would bring the flexibility that permissions provide for reasoning about dependent objects to traditional program verification. Meanwhile, our full permissions are remarkably similar to the "owner-as-modifier" paradigm used for verifying programs in Spec# and the JML, suggesting that permissions could largely supersede ownership in program verification. This in particular since transferring permissions between objects is simpler than transferring ownership. This is because permission transfer is a local operation between the provider and the consumer of a permission, while ownership is a globally visible property whose modifications can consequently have global impact (Müller and Rudich, 2007). We believe that separation guarantees between different pieces of owned data transfer to permissions: when data is only modified through full permissions then the state information associated with share permissions cannot have changed.

Third, the techniques developed for tracking permissions in Plural might be useful to automate reasoning in separation logic. Fractional permissions have been combined with separation logic (Bornat et al., 2005), but separation logic is undecidable and often relies on manual proofs. Static analyses have been successful in deriving separation logic constraints (Magill et al., 2006; Calcagno et al., 2009), and we believe that our permission inference could be used to include fractional permissions into these approaches.

Finally, it appears that typestates can be fruitfully combined with other areas of research. For instance, software model checking may scale to larger programs when using typestates to abstract the behavior of software components. Model checking may subsequently be used to check higher-level properties than protocol compliance of API objects. Test case generation may also benefit: it is currently very difficult to test higher-level classes automatically that rely on input data in a particular "state". Typestate-based protocols may simplify automatic test data generation in these situations. Finally, it may be possible to trace high-level business rules, such as "an account cannot be overdrawn," into code using typestates, which would establish a better connection between requirements and code.

## 9.6 Summary

This dissertation aims to provide comprehensive help to developers in using and providing APIs with associated usage protocols based on typestates (Strom and Yemini, 1986). The dissertation proposes the novel techniques of state refinement and access permissions to overcome challenges in expressiveness, handling of subtyping and inheritance, and aliasing flexibility in previous protocol checking approaches. It provides a

*static* technique for sound and modular protocol checking in conventional object-oriented software that can be automated in practical tools for interactive use.

The dissertation describes a prototype tool, Plural, that plugs into the Eclipse development environment to check protocol compliance in conventional Java code. Plural can check protocols in our case studies with sufficient performance and precision. Plural is open-source and uses Java 5 annotations to capture protocol-related design intent directly in Java source code.

Case studies with APIs from the Java standard library indicate sufficient expressiveness of the presented approach for practical protocols. Case studies with open-source programs indicate high precision and low developer overhead when using the presented approach for checking protocol compliance in off-the-shelf software. The additional protocol-related design intent of the presented has the extent and feel of conventional typing information.

This dissertation shows that state refinement and access permissions capture real API protocols, can be used for sound modular protocol enforcement, can be automated in tools for developers, and can check protocol compliance in practical object-oriented software. These results suggest that the presented approach could be used in practical software development, and that permissions could form the basis of future type systems for practical programming languages.

# Bibliography


A. Aiken, J. S. Foster, J. Kodumal, and T. Terauchi. Checking and inferring local non-aliasing. In *ACM Conference on Programming Language Design and Implementation*, pages 129–140, June 2003. ISBN 1-58113-662-5. doi: http://doi.acm.org/10.1145/781131.781146. 8.1, 8.1.2

J. Aldrich, V. Kostadinov, and C. Chambers. Alias annotations for program understanding. In *ACM Conference on Object-Oriented Programming, Systems, Languages & Applications*, pages 311–330, Nov. 2002. 8.1.1

R. Allen and D. Garlan. A formal basis for architectural connection. *ACM Transactions on Software Engineering and Methodology*, 6(3):213–249, July 1997. 8.3

J.-M. Andreoli. Logic programming with focusing proofs in linear logic. *Journal of Logic and Computation*, 2(3):197–347, 1992. 5.1.3

T. Ball and S. K. Rajamani. Automatically validating temporal safety properties of interfaces. In *Proceedings of the Eighth SPIN Workshop*, pages 101–122, May 2001. 8.1.2

M. Barnett, R. DeLine, M. Fähndrich, K. R. M. Leino, and W. Schulte. Verification of object-oriented programs with invariants. *Journal of Object Technology*, 3(6):27–56, June 2004. 1.1.2, 1.2.4, 7.2.2, 7.3.1, 7.4.3, 8.1, 8.1.1, 8.5, 9.2.4, 9.5

N. E. Beckman, K. Bierhoff, and J. Aldrich. Verifying correct usage of Atomic blocks and typestate. In *ACM Conference on Object-Oriented Programming, Systems, Languages & Applications*, pages 227–244, Oct. 2008. 4, 1.3, 3.2, 5, 6, 9.2.4, 9.2.6, 9.5

K. Bierhoff. Iterator specification with typestates. In *5th International Workshop on Specification and Verification of Component-Based Systems*, pages 79–82. ACM Press, Nov. 2006. 1.2.1, 1.2.3, 3.1, 3.1.3, 3.3, 6.4.3, 6.4.5, 7.1.2, 2, 3, 9.3.3

K. Bierhoff. Personal communication, Jan. 2009. 1.3

K. Bierhoff and J. Aldrich. Lightweight object specification with typestates. In *Joint European Software Engineering Conference and ACM Symposium on the Foundations of Software Engineering*, pages 217–226, Sept. 2005. 1.1.2, 1.1.2, 1.1.2, 1.2.1, 3.1.3, 3.1.6, 3.2, 3.2.1, 4.1.6, 6.4.3, 8.1, 8.2, 9.2.4

K. Bierhoff and J. Aldrich. Modular typestate verification of aliased objects. Technical Report CMU-ISRI-07-105, Carnegie Mellon University, Mar. 2007a. URL `http://reports-archive.adm.cs.cmu.edu/anon/isri2007/CMU-ISRI-07-105.pdf`. 1.1.2, 1.4, 2.2.2, 4, 4.2.6, 9.1.2, 9.4

K. Bierhoff and J. Aldrich. Modular typestate checking of aliased objects. In *ACM Conference on Object-Oriented Programming, Systems, Languages & Applications*, pages 301–320, Oct. 2007b. 1.1.2, 1.2, 1.2.3, 4, 6.3.7, 7.2.2, 8.1, 8.1.1

K. Bierhoff and J. Aldrich. Permissions to specify the composite design pattern. In *7th International Workshop on Specification and Verification of Component-Based Systems*, Nov. 2008. 1.3, 6.3.6, 7.3.2, 7.4.2, 9.5

K. Bierhoff, N. E. Beckman, and J. Aldrich. Practical API protocol checking with Access Permissions. In *European Conference on Object-Oriented Programming*. To appear, July 2009. 1.1.2, 7

E. Bodden. Personal communication, Feb. 2009. 7.3.3

E. Bodden, P. Lam, and L. Hendren. Finding programming errors earlier by evaluating runtime monitors ahead-of-time. In *ACM Symposium on the Foundations of Software Engineering*, pages 36–47, Nov. 2008. (document), 5, 6.4.5, 7.3, 7, 7.3.3, 7.3.3, 7.3.3, 7.4, 7.3.3, 8.1, 8.1.2, 8.2, 9.1.4, 9.1.5

E. Bodden, F. Chen, and G. Rosu. Dependent advice: A general approach to optimizing history-based aspects. In *ACM International Conference on Aspect-Oriented Software Development*, pages 3–14, Mar. 2009. doi: http://doi.acm.org/10.1145/1509239.1509243. 7.3.3, 8.2

R. Bornat, C. Calcagno, P. O'Hearn, and M. Parkinson. Permission accounting in separation logic. In *ACM Symposium on Principles of Programming Languages*, pages 259–270, Jan. 2005. doi: http://doi.acm.org/10.1145/1047659.1040327. 1.3, 5, 8.4, 8.5, 9.5

J. Boyland. Checking interference with fractional permissions. In *International Symposium on Static Analysis*, pages 55–72. Springer, 2003. 1.2.3, 5, 1.2.3, 1.3, 3.1.3, 3.1.4, 3.1.4, 3.3, 4.1.3, 5, 8.1.1, 8.1.1, 8.4

J. Boyland, W. Retert, and Y. Zhao. Iterators can be independent "from" their collections. In *International Workshop on Aliasing, Confinement, and Ownership in object-oriented programming*, July 2007. 7.1.2, 4, 7.3.2, 9.3.2

J. T. Boyland and W. Retert. Connecting effects and uniqueness with adoption. In *ACM Symposium on Principles of Programming Languages*, pages 283–295, Jan. 2005. 6.4.5, 8.1.1, 8.1.1

S. Butkevich, M. Renedo, G. Baumgartner, and M. Young. Compiler and tool support for debugging object protocols. In *ACM Symposium on the Foundations of Software Engineering*, pages 50–59, 2000. 8.2

C. Calcagno, D. Distefano, P. O'Hearn, and H. Yang. Compositional shape analysis by means of bi-abduction. In *ACM Symposium on Principles of Programming Languages*, pages 289–300. ACM, Jan. 2009. ISBN 978-1-60558-379-2. doi: http://doi.acm.org/10.1145/1480881.1480917. 8.5, 9.5

I. Cervesato, J. S. Hodas, and F. Pfenning. Efficient resource management for linear logic proof search. *Theoretical Computer Science*, 232:133–163, Feb. 2000. 5, 5.1

S. Chaki, E. Clarke, A. Groce, S. Jha, and H. Veith. Modular verification of software components in C. In *International Conference on Software Engineering*, pages 385–395, May 2003. 8.1.2

B. Chin, S. Markstrum, and T. Millstein. Semantic type qualifiers. In *ACM Conference on Programming Language Design and Implementation*, pages 85–95, 2005a. 8.1

W.-N. Chin, S.-C. Khoo, S. Qin, C. Popeea, and H. H. Nguyen. Verifying safety policies with size properties and alias controls. In *International Conference on Software Engineering*, pages 186–195, May 2005b. ISBN 1-59593-963-2. doi: http://doi.acm.org/10.1145/1062455.1062500. 4.1.1, 8.1, 8.1.1, 8.1.1

K. Crary, D. Walker, and G. Morrisett. Typed memory management in a calculus of capabilities. In *ACM Symposium on Principles of Programming Languages*, pages 262–275, 1999. ISBN 1-58113-095-3. doi: http://doi.acm.org/10.1145/292540.292564. 8.1.1

M. Das, S. Lerner, and M. Seigle. ESP: path-sensitive program verification in polynomial time. In *ACM Conference on Programming Language Design and Implementation*, pages 57–68, 2002. ISBN 1-58113-463-0. doi: http://doi.acm.org/10.1145/512529.512538. 8.1.2

L. de Alfaro and T. A. Henzinger. Interface automata. In *ACM Symposium on the Foundations of Software Engineering*, pages 109–120, Sept. 2001. 8.3

M. Degen, P. Thiemann, and S. Wehr. Tracking linear and affine resources with Java(X). In *European Conference on Object-Oriented Programming*, pages 550–574. Springer, Aug. 2007. 7.2.5, 8.1, 8.1.1, 8.1.1, 8.1.1

R. DeLine and M. Fähndrich. Enforcing high-level protocols in low-level software. In *ACM Conference on Programming Language Design and Implementation*, pages 59–69, 2001. URL `citeseer.ist.psu.edu/fahndrich01enforcing.html`. 1.1.2, 3.2, 3.2.1, 4.1.4, 8.1, 8.1.1

R. DeLine and M. Fähndrich. The Fugue protocol checker: Is your software baroque? Technical Report MSR-TR-2004-07, Microsoft Research, 2004a. 8.1.1

R. DeLine and M. Fähndrich. Typestates for objects. In *European Conference on Object-Oriented Programming*, pages 465–490. Springer, 2004b. (document), 1.1.2, 1.1.2, 1.2.3, 3.2, 3.2.1, 3.2.1, 3.2.2, 3.3, 4.2.2, 6.3.7, 8.1, 8.1.1, 8.1.1, 8.1.1, 8.1.1, 8.5

K. K. Dhara and G. T. Leavens. Forcing behavioral subtyping through specification inheritance. In *International Conference of Software Engineering*, pages 258–267, 1996. 8.5

W. Dietl and P. Müller. Universes: Lightweight ownership for JML. *Journal of Object Technology*, 4(8): 5–32, 2005. URL `http://www.jot.fm/issues/issues200510/article1`. 7.1.2, 9.5

J. Dunfield and F. Pfenning. Tridirectional typechecking. In *ACM Symposium on Principles of Programming Languages*, pages 281–292, 2004. URL `citeseer.ist.psu.edu/dunfield04tridirectional.html`. 1.2.1, 6.3.4, 6.4.4

M. B. Dwyer, A. Kinneer, and S. Elbaum. Adaptive online program analysis. In *International Conference on Software Engineering*, pages 220–229. IEEE Computer Society, May 2007. 8.1, 8.2

M. Fähndrich and R. DeLine. Adoption and focus: Practical linear types for imperative programming. In *ACM Conference on Programming Language Design and Implementation*, pages 13–24, June 2002. URL `citeseer.ist.psu.edu/fahndrich02adoption.html`. 4.2.2, 4.2.3, 8.1.1

M. Fähndrich and S. Xia. Establishing object invariants with delayed types. In *ACM Conference on Object-Oriented Programming, Systems, Languages & Applications*, pages 337–350, Oct. 2007. ISBN 978-1-59593-786-5. doi: http://doi.acm.org/10.1145/1297027.1297052. 6.4.6, 9.5

G. G. Fairbanks. *Design Fragments*. PhD thesis, Carnegie Mellon University, School of Computer Science, Apr. 2007. URL `http://reports-archive.adm.cs.cmu.edu/anon/isri2007/abstracts/07-108.html`. 8.1, 9.2.2

P. Ferrara, F. Logozzo, and M. Fähndrich. Safer unsafe code for .NET. In *ACM Conference on Object-Oriented Programming, Systems, Languages & Applications*, pages 329–346, 2008. 6.4.6

S. Fink, E. Yahav, N. Dor, G. Ramalingam, and E. Geay. Effective typestate verification in the presence of aliasing. In *ACM International Symposium on Software Testing and Analysis*, pages 133–144, July 2006. ISBN 1-59593-263-1. 8.1, 8.1.2

C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. Saxe, and R. Stata. Extended static checking for Java. In *ACM Conference on Programming Language Design and Implementation*, pages 234–245, May 2002. 1.1.2, 6.4.6, 8.5

J. S. Foster, T. Terauchi, and A. Aiken. Flow-sensitive type qualifiers. In *ACM Conference on Programming Language Design and Implementation*, pages 1–12, 2002. URL `citeseer.ist.psu.edu/foster02flowsensitive.html`. 6.4.4, 8.1, 8.1.1, 8.1.2

D. Giannakopoulou, C. S. Păsăreanu, and J. M. Cobleigh. Assume-guarantee verification of source code with design-level assumptions. In *International Conference on Software Engineering*, pages 211–220, May 2004. 1.2.3, 8.1.2, 8.3

J.-Y. Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987. 3.1.3, 5, 8.5

S. Gulwani and A. Tiwari. Constraint-based approach for analysis of hybrid systems. In *Computer Aided Verification*, pages 190–203, July 2008. 9.2.5

S. Gulwani, S. Srivastava, and R. Venkatesan. Program analysis as constraint solving. In *ACM Conference on Programming Language Design and Implementation*, pages 281–292, 2008. ISBN 978-1-59593-860-2. doi: http://doi.acm.org/10.1145/1375581.1375616. 9.2.5

J. Guttag, J. Horning, and J. Wing. The larch family of specification languages. *IEEE Software*, 2(5):24–36, Sept. 1985. 8.5

C. Haack and C. Hurlin. Resource usage protocols for iterators. In *International Workshop on Aliasing, Confinement and Ownership*, July 2008. 6.4.5, 7.1.2, 7.1.2

B. Hackett, M. Das, D. Wang, and Z. Yang. Modular checking for buffer overflows in the large. In *28th International Conference on Software Engineering*, pages 232–241. ACM, 2006. ISBN 1-59593-375-1. doi: http://doi.acm.org/10.1145/1134285.1134319. 9.5

S. Hallem, B. Chelf, Y. Xie, and D. Engler. A system and language for building system-specific, static analyses. In *ACM Conference on Programming Language Design and Implementation*, pages 69–82, 2002. ISBN 1-58113-463-0. doi: http://doi.acm.org/10.1145/512529.512539. 8.1.2

D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8: 231–274, 1987. 1.2, 1.2.1, 3.1.3, 3.1.6, 8.2

D. Harel and A. Naamad. The statemate semantics of statecharts. *ACM Transactions on Software Engineering Methodology*, 5(4):293–333, October 1996. 8.2

T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *ACM Symposium on Principles of Programming Languages*, pages 58–70, 2002. 8.1, 8.1.2

T. A. Henzinger, R. Jhala, and R. Majumdar. Permissive interfaces. In *Joint European Software Engineering Conference and ACM Symposium on the Foundations of Software Engineering*, pages 31–40, Sept. 2005. 8.6

T. Hoare. *Communicating Sequential Processes*. Prentice Hall International, 1985. 8.3

G. Hughes and T. Bultan. Interface grammars for modular software model checking. In *ACM Int. Symposium on Software Testing and Analysis*, pages 39–49. ACM Press, July 2007. ISBN 978-1-59593-734-6. doi: http://doi.acm.org/10.1145/1273463.1273471. 8.1.2

A. Igarashi and N. Kobayashi. Resource usage analysis. In *ACM Symposium on Principles of Programming Languages*, pages 331–342, Jan. 2002. ISBN 1-58113-450-9. doi: http://doi.acm.org/10.1145/503272.503303. 8.1, 8.1.1

A. Igarashi, B. Pierce, and P. Wadler. Featherweight Java: A minimal core calculus for Java and GJ. In *ACM Conference on Object-Oriented Programming, Systems, Languages & Applications*, pages 132–146, 1999. URL `citeseer.ist.psu.edu/igarashi99featherweight.html`. 4.1.1, 5.1.1

B. Jacobs and F. Piessens. The VeriFast program verifier. Technical Report CW-520, Department of Computer Science, Katholieke Universiteit Leuven, Aug. 2008. 7.3.2, 8.4

J. Jaffar and J.-L. Lassez. Constraint logic programming. In *ACM Symposium on Principles of Programming Languages*, pages 111–119, Jan. 1987. 5.1

C. Jaspan and J. Aldrich. Checking framework interactions with relationships. In *European Conference on Object-Oriented Programming*. To appear, July 2009. 9.2.2

N. Krishnaswami. Reasoning about iterators with separation logic. In *5th International Workshop on Specification and Verification of Component-Based Systems*, pages 83–86. ACM Press, Nov. 2006. 6.4.5, 7.1.2

V. Kuncak, P. Lam, K. Zee, and M. Rinard. Modular pluggable analyses for data structure consistency. *IEEE Transactions on Software Engineering*, 32(12):988–1005, Dec. 2006. URL `http://dx.doi.org/10.1109/TSE.2006.125`. 8.1, 8.1.1

G. T. Leavens, A. L. Baker, and C. Ruby. JML: A notation for detailed design. In H. Kilov, B. Rumpe, and I. Simmonds, editors, *Behavioral Specifications of Businesses and Systems*, pages 175–188. Kluwer Academic Publishers, Boston, 1999. 3.2.1, 6.4.3, 7.2.2, 7.3.1, 8.5

K. R. M. Leino. Data groups: Specifying the modification of extended state. In *ACM Conference on Object-Oriented Programming, Systems, Languages & Applications*, pages 144–153, Oct. 1998. 4.2.4, 8.1.1

K. R. M. Leino and P. Müller. A basis for verifying multi-threaded programs. In *European Symposium on Programming*, pages 378–393. Springer, Mar. 2009. 5, 6.3.6, 8.4, 9.5

P. Lincoln and A. Scedrov. First-order linear logic without modalities is NEXPTIME-hard. *Theoretical Computer Science*, 135:139–154, 1994. 4.2.5, 5

B. H. Liskov and J. M. Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems*, 16(6):1811–1841, Nov. 1994. 1.1.2, 1.2.2, 4.1.6, 8.5, 9.3.3

B. Livshits and T. Zimmermann. DynaMine: Finding common error patterns by mining software revision histories. In *Joint European Software Engineering Conference and ACM Symposium on the Foundations of Software Engineering*, pages 296–305, Sept. 2005. 8.1.2, 8.6

J. Magee, J. Kramer, and D. Giannakopoulou. Behaviour analysis of software architectures. In *Working IFIP Conference on Software Architecture*, pages 35–50, Feb. 1999. 8.3

S. Magill, A. Nanevski, E. Clarke, and P. Lee. Inferring invariants in separation logic for imperative list-processing programs. In *Workshop on Semantics, Program Analysis, and Computing Environments for Memory Management*, Jan. 2006. 8.5, 9.5

Y. Mandelbaum, D. Walker, and R. Harper. An effective theory of type refinements. In *ACM International Conference on Functional Programming*, pages 213–225, 2003. 4.1.1, 8.1, 8.1.1

B. Meyer. *Eiffel: The Language*. Prentice Hall, 1992. 8.5

G. Morrisett, A. Ahmed, and M. Fluet. L3: A linear language with locations. In *International Conference on Typed Lambda Calculi and Applications*, pages 293–307. Springer, Apr. 2005. 8.1.1

P. Müller and A. Rudich. Ownership transfer in Universe types. In *ACM Conference on Object-Oriented Programming, Systems, Languages & Applications*, pages 461–478, Oct. 2007. ISBN 978-1-59593-786-5. doi: http://doi.acm.org/10.1145/1297027.1297061. 9.5

N. Naeem and O. Lhoták. Typestate-like analysis of multiple interacting objects. In *ACM Conference on Object-Oriented Programming, Systems, Languages & Applications*, pages 347–366, Oct. 2008. 5, 1.2.4, 6.4.5, 6.5.5, 7.3, 7.3.3, 7.3.3, 7.3.3, 7.3.3, 8.1, 8.1.2, 9.1.4, 9.1.5

M. G. Nanda, C. Grothoff, and S. Chandra. Deriving object typestates in the presence of inter-object references. In *ACM Conference on Object-Oriented Programming, Systems, Languages & Applications*, pages 77–96, Oct. 2005. ISBN 1-59593-031-0. doi: http://doi.acm.org/10.1145/1094811.1094818. 8.6

B. C. Neuman and T. Ts'o. Kerberos: An authentication service for computer networks. *IEEE Communications Magazine*, 32(9):33–38, Sept. 1994. 9.5

F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer, 1999. 6.2.2, 6.3.3, 8.4

T. Nipkow. Linear quantifier elimination. In *International Joint Conference on Automated Reasoning*, pages 18–33. Springer, 2008. 5.2.2

M. J. Parkinson and G. M. Bierman. Separation logic, abstraction and inheritance. In *ACM Symposium on Principles of Programming Languages*, pages 75–86, Jan. 2008. 8.5

D. E. Perry. The Inscape environment. In *International Conference on Software Engineering*, pages 2–11. ACM Press, 1989. ISBN 0-8186-1941-4. doi: http://doi.acm.org/10.1145/74587.74588. 8.1

B. C. Pierce. *Types and Programming Languages*. MIT Press, Cambridge, 2002. 6.3.3

X. Qi and A. C. Myers. Masked types for sound object initialization. In *ACM Symposium on Principles of Programming Languages*, pages 53–65, Jan. 2009. ISBN 978-1-60558-379-2. doi: http://doi.acm.org/10.1145/1480881.1480890. 6.4.6, 9.5

G. Ramalingam, A. Warshavsky, J. Field, D. Goyal, and M. Sagiv. Deriving specialized program analyses for certifying component-client conformance. In *ACM Conference on Programming Language Design and Implementation*, pages 83–94, 2002. ISBN 1-58113-463-0. doi: http://doi.acm.org/10.1145/512529.512540. 3.1.4

J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *IEEE Symposium on Logic in Computer Science*, pages 55–74, 2002. 8.5

J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, 2nd edition, 2004. 9.2.4

A. Schrijver. *Theory of Linear and Integer Programming*. Wiley, July 1998. 2, 5.2.2

F. Smith, D. Walker, and G. Morrisett. Alias types. In *European Symposium on Programming*, pages 366–381. Springer, 2000. 8.1.1

R. E. Strom and S. Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Transactions on Software Engineering*, 12:157–171, 1986. (document), 1.1.1, 1.1.2, 8.1, 9.6

D. F. Sutherland. *The Code of Many Colors: Semi-automated Reasoning about Multi-Thread Policy for Java*. PhD thesis, Carnegie Mellon University, School of Computer Science, May 2008. URL `http://reports-archive.adm.cs.cmu.edu/anon/isr2008/abstracts/08-112.html`. 7.3.2

G. Tan, X. Ou, and D. Walker. Enforcing resource usage protocols via scoped methods. In *International Workshop on Foundations of Object-Oriented Languages*, 2003. 8.1, 8.1.1

T. Terauchi. Checking race freedom via linear programming. In *ACM Conference on Programming Language Design and Implementation*, pages 1–10, June 2008. 5.1.2, 6.3.3, 6.3.3, 8.4, 9.2.5

T. Terauchi and A. Aiken. Witnessing side–effects. In *ACM International Conference on Functional Programming*, pages 105–115, 2005. ISBN 1-59593-064-7. doi: http://doi.acm.org/10.1145/1086365.1086379. 8.4

T. Terauchi and A. Aiken. A capability calculus for concurrency and determinism. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 30(5):1–30, Aug. 2008. ISSN 0164-0925. doi: http://doi.acm.org/10.1145/1387673.1387676. 1.3, 5, 5.1.2, 6.3.3, 8.4

P. Wadler. Linear types can change the world! In *Working Conference on Programming Concepts and Methods*, pages 347–359. North Holland, 1990. 1.2.3, 4.1.3, 8.1.1

W. A. Wulf, R. L. London, and M. Shaw. An introduction to the construction and verification of Alphard programs. *IEEE Transactions on Software Engineering*, 2(4):253–265, Dec. 1976. 8.5

Y. Zhao. *Concurrency Analysis Based on Fractional Permission System*. PhD thesis, University of Wisconsin-Milwaukee, Aug. 2007. 4.2.5, 8.4